Incremental View Maintenance For Collection Programming*

Christoph Koch
EPFL
christoph.koch@epfl.ch

Daniel Lupei*
EPFL
daniel.lupei@epfl.ch

Val Tannen
University of Pennsylvania
val@cis.upenn.edu

ABSTRACT

In the context of incremental view maintenance (IVM), *delta* query derivation is an essential technique for speeding up the processing of large, dynamic datasets. The goal is to generate delta queries that, given a small change in the input, can update the materialized view more efficiently than via recomputation.

In this work we propose the first solution for the efficient incrementalization of positive nested relational calculus (NRC^+) on bags (with integer multiplicities). More precisely, we model the cost of NRC^+ operators and classify queries as efficiently incrementalizable if their delta has a strictly lower cost than full re-evaluation. Then, we identify IncNRC^+, a large fragment of NRC^+ that is efficiently incrementalizable and we provide a semantics-preserving translation that takes any NRC^+ query to a collection of IncNRC^+ queries. Furthermore, we prove that incremental maintenance for NRC^+ is within the complexity class NC_0 and we showcase how *recursive* IVM, a technique that has provided significant speedups over traditional IVM in the case of flat queries [25], can also be applied to IncNRC^+ .

1. INTRODUCTION

Large-scale collection processing in frameworks such as Spark [41] or LINQ [33] can greatly benefit from incremental maintenance in order to minimize query latency in the face of updates. These frameworks provide collection abstractions equivalent to nested relational operators that are embarrassingly parallelizable. Also, they can be aggressively optimized using powerful algebraic laws. Language-integrated querying makes use of this algebraic framework to turn declarative collection processing queries into efficient nested calculus expressions.

Incremental view maintenance (IVM) by static query rewriting (a.k.a. delta query derivation) has proven to be a highly useful and, for instance in the context of data warehouse loading, an indispensable feature of many commercial data management systems. With delta processing, the results of a query are incrementally maintained by a delta query that, given the original input and an incremental update, computes the corresponding change of the output. Query execution can thus be staged into an offline phase for running the query over an initial database and materializing the result, followed by an online phase in which the delta query is evaluated and its result applied to the materialized view upon receiving updates. This execution model means that one can do as much as possible once and for all before any updates are first seen, rather than process the entire input every time data changes.

Delta processing is worthwhile only if delta query evaluation is much cheaper than full re-computation. In many cases deltas are actually asymptotically faster – for instance, filtering the input based on some predicate takes linear time, whereas the corresponding delta query does not need to access the database but only considers the incremental update, and thus runs in time proportional to the size of the update (in practice, usually constant time).

The benefits of incremental maintenance can be amplified if one applies it recursively [24] – the evaluation of delta queries themselves can be sped up by materializing and incrementally maintaining their results using second-order delta-queries (deltas of the delta queries). One can build a hierarchy of delta queries, where the deltas at each level are used to maintain the materialization of deltas above them, all the way up to the original query. This approach of *higher-order* delta derivation (a.k.a. recursive IVM) admits a complexity-theoretic separation between re-evaluation and incremental maintenance of positive relational queries with aggregates (RA^+_Σ) [24], and outperforms classical IVM by many orders of magnitude [25]. Unfortunately, the techniques described above target only flat relational queries and as such cannot be used to enable incremental maintenance for collection processing engines.

In this work we address the problem of delta processing for positive nested-relational calculus on bags (NRC^+). Specifically, we consider deltas for updates that are applied to the input relations via a generalized bag union $\uplus$ (which sums up multiplicities), where tuples have integer multiplicities in order to support both insertions and deletions. We formally define what it means for a nested update to be *incremental* and a NRC^+ query to be *efficiently* incrementalizable, and we propose the first solution for the efficient incremental maintenance of NRC^+ queries.

This work was supported by ERC grant 279804 and NSF grants IIS-1217798 and IIS-1302212.
*Corresponding author.

PODS'16, June 26-July 01, 2016, San Francisco, CA, USA
 ISBN 978-1-4503-4191-2/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2902251.2902286

We say that a query is efficiently incrementalizable if its delta has a lower cost than recomputation. We define cost domains equipped with partial orders for every nested type in NRC^+ and determine cost functions for the constructs of NRC^+ based on their semantics and a lazy evaluation strategy. The cost domains that we use attach a cardinality estimate to each nesting level of a bag, where the cardinality of a nesting level is defined as the maximum cardinality of all the bags with the same nesting level. For example, to the nested bag $\{\{a\},\{b\},\{c,d\}\}$ we associate a cost value of $3\{2\}$, since the top bag has 3 elements and the inner bags have a maximum cardinality of 2. This choice of cost domains was motivated by the fact that data may be distributed unevenly across the nesting levels of a bag, while one can write queries that operate just on a particular nested level of the input. Even though our cost model makes several conservative approximations, it is still precise enough to separate incremental maintenance from re-evaluation for a large fragment of NRC^+.

We efficiently incrementalize NRC^+ in two steps. We first establish IncNRC^+, the largest fragment for which we can derive efficient deltas. Then, for queries in $\text{NRC}^+ \setminus \text{IncNRC}^+$, we provide a semantics preserving translation into a collection of IncNRC^+ queries on a differently represented database.

For IncNRC^+ we leverage the fact that our delta transformation is *closed* (i.e. maps to the same query language) and illustrate how to further optimize delta processing using recursive IVM: if the delta of an IncNRC^+ query still depends on the database, it follows that it can be partially evaluated and efficiently maintained using a higher-order delta. We show that for any IncNRC^+ query there are only a finite number of higher-order delta derivations possible before the resulting expressions no longer depend on the database (but are purely functions of the update), and thus no longer require maintenance.

The only queries that fall outside IncNRC^+ are those that use the singleton bag constructor $\mathbf{sng}(e)$, where e depends on the database. This is supported by the intuition that in NRC^+ we do not have an efficient way to modify $\mathbf{sng}(e)$ into $\mathbf{sng}(e \uplus \Delta e)$, without first removing $\mathbf{sng}(e)$ from the view and then adding $\mathbf{sng}(e \uplus \Delta e)$, which amounts to recomputation. The challenge of efficiently applying updates to inner bags, a.k.a. *deep* updates, does not lie in designing an operator that navigates the structure of a nested object and applies the update to the right inner bag, but doing so while providing useful re-writing rules wrt. the other language constructs, which can be used to derive efficient delta queries. Previous approaches to incremental maintenance of nested views have either ignored the issue of deep updates [15], handled it by triggering recomputation of nested bags [32] or defaulted to change propagation [34, 22].

We address the problem of efficiently incrementalizing $\mathbf{sng}(e)$ with *shredding*, a semantics-preserving transformation that replaces the inner bag introduced by $\mathbf{sng}(e)$ with a label l and separately maintains the mapping between l and its defining query e. Therefore, deep updates can be applied by simply modifying the label definition corresponding to the inner bag being updated. As such, the problem of incrementalizing NRC^+ queries is reduced to that of incrementalizing the collection of IncNRC^+ queries resulting from the shredding transformation. Furthermore, based on this reduction we also show that, analogous to the flat relational case [24], incremental processing of NRC^+ queries is in a strictly lower complexity class than re-evaluation (NC_0 vs. TC_0).

The idea of encoding inner bags by fresh indices/labels and then keeping track of the mapping between the labels and the contents of those bags has been studied before in the literature in various contexts [9, 23, 10, 28, 38, 18]. However we are, to the best of our knowledge, the first to propose a generic and compositional shredding transformation for solving the problem of efficient IVM for NRC^+ queries. The compositional nature of our solution is essential for applications where nested data is exchanged between several layers of the system.

We summarize our contributions as follows:

- We define the notions of *incremental* nested update and *efficient incrementalization* of nested queries, based on cost domains and a cost interpretation over NRC^+'s constructs.
- We provide the first solution for the efficient incrementalization of positive nested-relational calculus (NRC^+).
- We show how delta processing of nested queries can be further optimized using recursive IVM [24].
- We show that incremental evaluation is in a strictly lower complexity class than re-computation (NC_0 vs. TC_0).

The rest of the paper is organized as follows. We first introduce our approach for the incrementalization of NRC^+ queries on a motivating example and formally define the variant of positive nested relational calculus that we use. The efficient delta processing of a large fragment of NRC^+ is discussed in Section 4 and in Section 5 we show how the full NRC^+ can be efficiently maintained. Finally, in Section 6 we review the related literature. Some of the proofs have been ommited for space reasons but can be found in the long version of this paper [26], along with additional examples/discussions.

2. MOTIVATING EXAMPLE

We follow the classical approach to incremental query evaluation, which is based on applying certain syntactic transformations called "delta rules" to the query expressions of interest (in [26] we revisit how delta processing works for the flat relational case). In the following, we give some intuition for the difficulties that arise in finding a delta rules approach to the problem of incremental computation on *nested* bag relations.

Notation. For a query Q and relation R, we denote by $Q[R]$ the fact that Q is defined in terms of input R. We will sometimes simply write Q, if R is obvious from the context.

EXAMPLE 1. *We consider the query* `related` *that computes for every movie in relation* $M(name, gen, dir)$ *a set of related movies which are either in the same genre gen or share the same artistic director dir. We define* `related` *in Spark*[1]*:*

```
case class Movie(name: String, gen: String, dir: String)
val M: RDD[Movie] = ...
val related = for(m <- M) yield (m.name, relB(m))
def relB(m: Movie) =
  for(m2 <- M if isRelated(m,m2)) yield m2.name
def isRelated(m: Movie, m2: Movie) =
  m.name != m2.name && (m.gen==m2.gen || m.dir==m2.dir)
```

[1]To improve the presentation we omitted Spark's boilerplate code.

where `RDD` *is Spark's collection type for distributed datasets,* `relB`(m) *computes the names of all the movies related to* m *and* `isRelated` *tests if two different movies are related by genre or director. We evaluate* `related` *on an example instance.*

M

name	*gen*	*dir*
Drive	*Drama*	*Refn*
Skyfall	*Action*	*Mendes*
Rush	*Action*	*Howard*

`related`$[M]$

name	$\{name\}$
Drive	$\{\}$
Skyfall	$\{Rush\}$
Rush	$\{Skyfall\}$

Now consider the outcome of updating M *with* ΔM *via bag union* $\uplus$*, where* ΔM *is a relation with the same schema as* M *and contains a single tuple* $\langle Jarhead, Drama, Mendes\rangle$.

$M \uplus \Delta M$

name	*gen*	*dir*
Drive	*Drama*	*Refn*
Skyfall	*Action*	*Mendes*
Rush	*Action*	*Howard*
Jarhead	*Drama*	*Mendes*

`related`$[M \uplus \Delta M]$

name	$\{name\}$
Drive	$\{Jarhead\}$
Skyfall	$\{Rush,\ Jarhead\}$
Rush	$\{Skyfall\ \}$
Jarhead	$\{Drive,\ Skyfall\}$

To incrementally update the result of `related` we design a set of delta rules that, when applied to the definition of `related`$[M]$, give us an expression $\delta($`related`$)[M, \Delta M]$ s.t.:

`related`$[M \uplus \Delta M]$ = `related`$[M] \uplus \delta($`related`$)[M, \Delta M]$.

For our example, in order to modify `related`$[M]$ into `related`$[M \uplus \Delta M]$, without completely replacing the existing tuples[2], one would have to add the movie Jarhead to the inner bag of related movies for Drive (same genre) and Skyfall (same director). However, our target language of Nested Relational Calculus (NRC) [6, 27, 11, 12] (with bag semantics, where tuples have integer multiplicities in order to support both insertions and deletions [29, 24]) is not equipped with the necessary constructs for expressing this kind of changes, and efficiently processing such 'deep' updates represents the main challenge in incrementally maintaining nested queries. Although update operations able to perform deep changes have been proposed in the literature [30], they lack the necessary re-write rules needed for a *closed* delta transformation, which is a prerequisite for recursive IVM.

In order to make inner bags accessible by 'deep' updates, we must first devise a naming scheme to address them. We have two options: i) we can either associate a label to each tuple in a bag and then identify an inner bag based on this label and the index of the tuple component that contains the bag, or ii) we can associate a label to each inner bag, and separately maintain a mapping between the label and the corresponding inner bag. In other words, labels can either identify the position of an inner bag within the nested value or serve as an alias for the contents of the inner bag. For example, given a value $X = \{\langle a, \{x_1, x_2\}\rangle, \langle b, \{x_3\}\rangle\}$, the first alternative decorates it with labels as follows: $\{l_1 \mapsto \langle a, \{x_1, x_2\}\rangle, l_2 \mapsto \langle b, \{x_3\}\rangle\}$, and then addresses the inner bags by $l_1.2$ and $l_2.2$. By contrast, the second approach creates the mappings $l_1 \mapsto \{x_1, x_2\}$ and $l_2 \mapsto \{x_3\}$, and then represents the original value as the flat bag $X^F = \{\langle a, l_1\rangle, \langle b, l_2\rangle\}$.

Even though both schemes faithfully represent the original nested value, we prefer the second one, a.k.a. *shredding* [9, 18], as it offers a couple of advantages. Firstly, it makes the contents of the inner bags conveniently accessible to updates via regular bag addition, without the need to introduce a custom update operation[3]. Secondly, since inner bags are represented by labels it also avoids duplicating their contents. For example, when computing the Cartesian product of X with some bag Y, one would normally create a copy of the tuples in X, along with their inner bags, for each element of Y. Moreover, any update of an inner bag from X would also have to be applied to every instance of that bag appearing in the output of $X \times Y$. By contrast, the second scheme computes the Cartesian product only between X^F and Y, while the mappings between labels and the contents of the inner bags remain untouched. Therefore, any update to an inner bag of X can be efficiently applied just by updating its corresponding mapping.

For operating over nested values represented in shredded form, we propose a semantics-preserving transformation that rewrites a query with nested output $Q[R]$ into a query Q^F returning the flat representation of the result, and a series of queries Q^Γ, computing the contents of its inner bags.

2.1 Incrementalizing `related`

We showcase our approach on the motivating example by first expressing it in NRC. The main constructs that we use are: i) the for-comprehension **for** x **in** Q_1 **where** $p(x)$ **union** $Q_2(x)$, which iterates over all the elements x from the output of query Q_1 that satisfy predicate $p(x)$ and unions together the results of each $Q_2(x)$, and ii) the singleton constructor **sng**(e), which creates a bag with the result of e as its only element.

$$\texttt{related} \equiv \textbf{for}\ m\ \textbf{in}\ M\ \textbf{union}\ \textbf{sng}(\langle m.name, \texttt{relB}(m)\rangle)$$

$$\texttt{relB}(m) \equiv \textbf{for}\ m_2\ \textbf{in}\ M\ \textbf{where}\ \texttt{isRelated}(m, m_2)\ \textbf{union}\ \textbf{sng}(m_2.name).$$

Next, we investigate the incrementalization of the constructs used by the `related` query in order to identify which one of them can lead to the problem of deep updates. The delta rule of the **for** construct is a natural generalization of the rule for Cartesian product in relational algebra[4]:

$$\begin{aligned}\delta(\textbf{for}\ x\ \textbf{in}\ Q_1\ \textbf{union}\ Q_2) = {} & \textbf{for}\ x\ \textbf{in}\ \delta(Q_1)\ \textbf{union}\ Q_2 \\ & \uplus\ \textbf{for}\ x\ \textbf{in}\ Q_1\ \textbf{union}\ \delta(Q_2) \\ & \uplus\ \textbf{for}\ x\ \textbf{in}\ \delta(Q_1)\ \textbf{union}\ \delta(Q_2)\end{aligned} \quad (1)$$

assuming we can derive corresponding deltas for Q_1 and Q_2.

If the **where** clause is also present, the same rule applies because we only consider the positive fragment of nested bag languages, for which predicates are not allowed to test expressions of bag type (the reasoning behind this decision is detailed in Appendix A). Therefore the predicates in the **where** clause can only be boolean combinations of comparisons involving base type expressions and these are not affected by updates of the database.

The difficulty arises when we try to design a delta rule for singleton, specifically, how to deal with **sng**(e) when e depends on some database relation. There is plainly no way in our calculus to express the change from **sng**(M) to **sng**$(M \uplus \Delta M)$ in an efficient manner, i.e., one that is proportional to the size of ΔM and not the size of the output. This

[2]Maintaining the result of `related` by completely replacing the affected tuples defeats the goal of making incremental computation more efficient than full re-evaluation, as these tuples could be arbitrarily large.

[3]The authors investigated this alternative and found it particularly challenging due to the complex ways in which this custom operation would interact with the existing constructs of the language.

[4]$\delta(e_1 \times e_2) = \delta(e_1) \times e_2\ \uplus\ e_1 \times \delta(e_2)\ \uplus\ \delta(e_1) \times \delta(e_2)$

is the same problem that we saw with the `related` example above. In Section 4 we will show that $\mathbf{sng}(e)$ is the only construct in our calculus whose efficient incrementalization relies on 'deep' updates.

2.2 Maintaining inner bags

In order to facilitate the maintenance of the bags produced by $\texttt{relB}(m)$, we associate to each one of them a label, and we store separately a mapping between the label and its bag. Then, for implementing updates to a nested bag, we can simply modify the definition of its associated label via bag union. We note that this strategy can be applied for enacting 'deep' changes to both nested materialized views as well as nested relations in the database.

Since the bags created by $\texttt{relB}(m)$ clearly depend on the variable m bound by the **for** construct, we also incorporate the values that m takes in the labels that replace them. The simplest way of doing so is to use labels that are pairs of indices and values, where the index uniquely identifies the inner query being replaced. In our running example, as we have just a single inner query, we only need one index ι.

The shredding of `related` yields two queries, $\texttt{related}^F$ producing a flat version of `related` with its inner bags replaced by labels, and $\texttt{related}^\Gamma$ that computes the value of a nested bag given a label *parameter* ℓ of the form $\langle \iota, m \rangle$

$$\texttt{related}^F \equiv \textbf{for } m \textbf{ in } M \textbf{ union } \mathbf{sng}(\langle m.name, \langle \iota, m \rangle \rangle)$$

$$\texttt{related}^\Gamma(\ell) \equiv \textbf{for } m_2 \textbf{ in } M \textbf{ where } \texttt{isRelated}(\ell.2, m_2) \textbf{ union } \mathbf{sng}(m_2.name)$$

The output of these queries on our running example is:

$\texttt{related}^F[M]$

name	ℓ
Drive	$\langle \iota, \langle$Drive,..$\rangle \rangle$
Skyfall	$\langle \iota, \langle$Skyfall,..$\rangle \rangle$
Rush	$\langle \iota, \langle$Rush,..$\rangle \rangle$

$\texttt{related}^\Gamma[M]$

ℓ	$\mapsto$	$\{name\}$
$\langle \iota, \langle$Drive,..$\rangle \rangle$	$\mapsto$	{}
$\langle \iota, \langle$Skyfall,..$\rangle \rangle$	$\mapsto$	{Rush}
$\langle \iota, \langle$Rush,..$\rangle \rangle$	$\mapsto$	{Skyfall}

Although in our example the generated queries are completely flat, this need not always be the case. In particular, in order to avoid expensive pre-/post-processing steps, one should perform shredding only down to the nesting level that is affected by the changes in the input.

Upon shredding, the strategy for incrementally maintaining `related` is to materialize and incrementally maintain $\texttt{related}^F$ and $\texttt{related}^\Gamma$, and then recover `related` from the results based on the following equivalence:

$$\texttt{related} = \textbf{for } r \textbf{ in } \texttt{related}^F \textbf{ union } \mathbf{sng}(\langle r.1, \texttt{related}^\Gamma(r.2) \rangle),$$

which holds since the values that m takes are incorporated in the labels ℓ, and $\texttt{related}^\Gamma(\ell)$ is essentially a rewriting of the subquery $\texttt{relB}(m)$.

We remark that, while being able to reconstruct `related` from $\texttt{related}^F$ and $\texttt{related}^\Gamma$ is important for proving the correctness of our transformation (see Section 5.3), it is not essential for representing the final result since the labels that appear in $\texttt{related}^F$ can simply be seen as references to the inner bags. We also note that even though $\texttt{related}^\Gamma$ is parameterized by ℓ, one can use standard domain maintenance techniques to materialize it since the relevant values of ℓ are ultimately those found in the tuples of $\texttt{related}^F$. Finally, in this example the labels are in bijection with the values over which m ranges, and hence, one could use those values themselves as labels. In general however we may have several nested subqueries that depend on the same variable m.

In the process of shredding queries we replace every subquery of a singleton construct that depends on the database with a label that does not. This is the case with the subquery $\texttt{relB}(m)$ in `related`, and we have a very simple delta rule for expressions that do not depend on the input bags: $\delta(\mathbf{sng}(\langle m.name, \langle \iota, m \rangle \rangle)) = \delta(\mathbf{sng}(m_2.name)) = \varnothing$. Therefore, applying delta rules such as (1) gives us:

$$\delta(\texttt{related}^F) = \textbf{for } m \textbf{ in } \Delta M \textbf{ union } \mathbf{sng}(\langle m.name, \langle \iota, m \rangle \rangle)$$

$$\delta(\texttt{related}^\Gamma)(\ell) = \textbf{for } m_2 \textbf{ in } \Delta M \textbf{ where } \texttt{isRelated}(\ell.2, m_2) \textbf{ union } \mathbf{sng}(m_2.name)$$

We shall prove in Section 4 that, for the class of queries to which $\texttt{related}^F$ and $\texttt{related}^\Gamma$ belong, the delta rules do indeed produce a proper update. We remark that since the domain of $\texttt{related}^\Gamma$ is determined by the labels in $\texttt{related}^F$, it may be extended by the $\delta(\texttt{related}^F)$ update. Thus, when updating the materialization of $\texttt{related}^\Gamma$ with the change produced by $\delta(\texttt{related}^\Gamma)$, one must also check whether each label in its domain has an associated definition, and if not initialize it accordingly.

Cost analysis. In the following we show that maintaining `related` incrementally is more efficient than its re-evaluation (for the general case see Section 4.2). Let us assume that M and ΔM have n and d tuples, respectively, including repetitions. From the expressions above it follows that the costs of computing the original queries ($\texttt{related}^F$ and $\texttt{related}^\Gamma(\ell)$) is proportional to the input, while their deltas cost $O(d)$.

As previously noted, $\texttt{related}[M \uplus \Delta M]$ can be recovered from:

$$\textbf{for } r \textbf{ in } \texttt{related}^F[M \uplus \Delta M] \textbf{ union } \mathbf{sng}(\langle r.1, \texttt{related}^\Gamma[M \uplus \Delta M](r.2) \rangle),$$

and by the properties of delta queries and one of the general equivalence laws of the NRC [6], this becomes $V \uplus W$ where

$$V = \textbf{for } r \textbf{ in } \texttt{related}^F[M] \textbf{ union } \mathbf{sng}(\langle r.1, \texttt{related}^\Gamma[M](r.2) \uplus \delta(\texttt{related}^\Gamma)(r.2) \rangle) \quad (2)$$

$$W = \textbf{for } r \textbf{ in } \delta(\texttt{related}^F) \textbf{ union } \mathbf{sng}(\langle r.1, \texttt{related}^\Gamma[M \uplus \Delta M](r.2) \rangle) \quad (3)$$

Even counting repetitions, we have $O(n)$ tuples in the materialization of $\texttt{related}^F[M]$ while the result of computing $\delta(\texttt{related}^F)$ has $O(d)$ tuples. From (2) the cost of computing V is $O(nd)$ and from (3) the cost of computing W is $O(d(n+d))$, where we assumed that unioning two already materialized bags takes time proportional to the smaller one, and looking up the definition of a label takes constant (amortized) time. Thus, the incremental computation of `related` costs $O(nd + d^2)$. For the costs of maintaining $\texttt{related}^F$ and $\texttt{related}^\Gamma$ we have $O(d)$ and $O(d(n+d))$, respectively, considering that initializing the new labels introduced by $\delta(\texttt{related}^F)$ takes $O(dn)$ and then updating all the definitions in $\texttt{related}^\Gamma$ takes $O((n+d)d)$ (which includes the cost of rehashing the labels in $\texttt{related}^\Gamma$ as may be required due to its increase in size). It follows that the overall cost of IVM is $O(nd + d^2)$ and when $n \gg d$, performing IVM is clearly much better than recomputing $\texttt{related}[M \uplus \Delta M]$ which costs $\Omega((n+d)^2)$ (in the step-counting model we have been using).

In the next sections we develop our approach in detail.

3. CALCULUS

We describe the version of the positive nested relational calculus (NRC$^+$) on bags that we use. Its types are:

$$A, B, C \;:=\; 1 \;\mid\; Base \;\mid\; A \times B \;\mid\; \mathbf{Bag}(C),$$

where $Base$ is the type of the database domain and 1 denotes the "unit" type (a.k.a. the type of the 0-ary tuple $\langle\rangle$). We also use $TBase$ to denote nested tuple types with components of only $Base$ type.

In order to capture all updates, i.e., both insertions and deletions, we use a generalized notion of bag where elements have (possibly negative) integer multiplicities and bag addition $\uplus$ sums multiplicities as integers. In addition, for every bag type we have an empty bag constructor $\varnothing$, as well as construct $\ominus(e)$ that negates the multiplicities of all the elements produced by e. We remark that, semantically, bag types along with empty bag $\varnothing$, bag addition $\uplus$ and bag minus $\ominus$ exhibit the structure of a commutative group. This implies that given any two query results Q_{old} and Q_{new}, there will always exist a value ΔQ s.t. $Q_{new} = Q_{old} \uplus \Delta Q$. This rich algebraic structure that bags exhibit is also the reason why we use a calculus with bag, as opposed to set semantics.

Typed calculus expressions $\Gamma; \Pi \vdash e : \mathbf{Bag}(B)$ have two sets of type assignments to variables $\Gamma = X_1{:}\mathbf{Bag}(C_1), \cdots, X_m{:}\mathbf{Bag}(C_m)$ and $\Pi = x_1{:}A_1, \cdots, x_n{:}A_n$, in order to distinguish between variables X_i defined via **let** bindings and which reference top level bags, and variables x_i which are introduced within **for** comprehensions and bind the inner elements of a bag. We maintain this distinction since in the process of shredding we will use the latter set to generate unique labels, identifying shredded bags (section 5.1).

The typing rules and semantics of NRC$^+$ are given in Figure 1, where R ranges over the relations in the database, X and x range over the variables in the contexts Γ and Π, respectively, **let** binds the result of e_1 to R and uses it in the evaluation of e_2, $\times$ performs Cartesian product of bags, **for** iteratively evaluates e_2 with x bound to every element of e_1 and then unions together all the resulting bags, **flatten** turns a bag of bags into just one bag by unioning the inner bags, **sng** places its input into a singleton bag and p stands for any predicate over tuples of primitive values. Compared to the standard formulation given in [6] we use a calculus version that is "delta-friendly" in that all expressions have bag type and more importantly most of its constructs are either linear or distributive wrt. to bag union, with the notable exception of $\mathbf{sng}(e)$. Therefore we have a bag (Cartesian) product construct instead of a pairing construct, we have a separate flattening construct, and we control carefully how singletons are constructed (note that we have four rules for singletons but they do not "overlap"). Finally, γ and ε are assignments of values to variables, and we denote their extension with a new assignment by $\gamma[X := v]$ and $\varepsilon[x := v]$, respectively. Throughout the presentation, we will omit such value assignments whenever they are not explicitly needed for resolving variable names.

Booleans are simulated by $\mathbf{Bag}(1)$, with the singleton bag $\mathbf{sng}(\langle\rangle)$ denoting *true* and the empty bag $\varnothing$ denoting *false*. Consequently, the return type of predicates $p(x)$ is also $\mathbf{Bag}(1)$. The "positivity" of the calculus is captured by the restriction put on (comparison) predicates $p(x)$ to only act on tuples of basic values since comparisons involving

$$\frac{\mathrm{Sch}(R){=}B}{R : \mathbf{Bag}(B)} \qquad \frac{\Gamma; \Pi \vdash e_1 : \mathbf{Bag}(C) \quad \Gamma, X{:}\mathbf{Bag}(C); \Pi \vdash e_2 : \mathbf{Bag}(B)}{\Gamma; \Pi \vdash \mathbf{let}\ X := e_1\ \mathbf{in}\ e_2 : \mathbf{Bag}(B)}$$

$$\frac{}{\Gamma, X : \mathbf{Bag}(C); \Pi \vdash X : \mathbf{Bag}(C)} \qquad \frac{}{\Gamma; \Pi, x : TBase \vdash p(x) : \mathbf{Bag}(1)}$$

$$\frac{}{\Gamma; \Pi, x{:}A \vdash \mathbf{sng}(x) : \mathbf{Bag}(A)} \qquad \frac{}{\mathbf{sng}(\langle\rangle) : \mathbf{Bag}(1)} \qquad \frac{}{\varnothing : \mathbf{Bag}(B)}$$

$$\frac{i = 1, 2}{\Gamma; \Pi, x{:}A_1 \times A_2 \vdash \mathbf{sng}(\pi_i(x)) : \mathbf{Bag}(A_i)} \qquad \frac{e : \mathbf{Bag}(B)}{\mathbf{sng}(e) : \mathbf{Bag}(\mathbf{Bag}(B))}$$

$$\frac{\Gamma; \Pi \vdash e_1 : \mathbf{Bag}(A) \quad \Gamma; \Pi, x{:}A \vdash e_2 : \mathbf{Bag}(B)}{\Gamma; \Pi \vdash \mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2 : \mathbf{Bag}(B)} \qquad \frac{e_{1,2} : \mathbf{Bag}(B)}{e_1 \uplus e_2 : \mathbf{Bag}(B)}$$

$$\frac{e_i : \mathbf{Bag}(B_i), i = 1, 2}{e_1 \times e_2 : \mathbf{Bag}(B_1 \times B_2)} \qquad \frac{e : \mathbf{Bag}(\mathbf{Bag}(B))}{\mathbf{flatten}(e) : \mathbf{Bag}(B)} \qquad \frac{e : \mathbf{Bag}(B)}{\ominus(e) : \mathbf{Bag}(B)}$$

$$[\![R]\!] = R \qquad [\![\mathbf{let}\ X := e_1\ \mathbf{in}\ e_2]\!]_{\gamma;\varepsilon} = [\![e_2]\!]_{\gamma[X := [\![e_1]\!]_{\gamma;\varepsilon}];\varepsilon}$$

$$[\![X]\!]_{\gamma;\varepsilon} = \gamma(X) \qquad [\![p(x)]\!]_{\gamma;\varepsilon} = \text{ if } p(\varepsilon(x)) \text{ then } \{\langle\rangle\} \text{ else } \{\}$$

$$[\![\mathbf{sng}(x)]\!]_{\gamma;\varepsilon} = \{\varepsilon(x)\} \qquad [\![\mathbf{sng}(\pi_i(x))]\!]_{\gamma;\varepsilon} = \{\pi_i(\varepsilon(x))\}$$

$$[\![\mathbf{sng}(e)]\!] = \{[\![e]\!]\} \qquad [\![\mathbf{flatten}(e)]\!] = \biguplus_{v \in [\![e]\!]} v$$

$$[\![\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2]\!]_{\gamma;\varepsilon} = \biguplus_{v \in [\![e_1]\!]_{\gamma;\varepsilon}} [\![e_2]\!]_{\gamma;\varepsilon[x:=v]}$$

$$[\![e_1 \times e_2]\!] = \biguplus_{v_1 \in [\![e_1]\!]} \biguplus_{v_2 \in [\![e_2]\!]} \{\langle v_1, v_2\rangle\} \qquad [\![\mathbf{sng}(\langle\rangle)]\!] = \{\langle\rangle\}$$

$$[\![\varnothing]\!] = \{\} \qquad [\![e_1 \uplus e_2]\!] = [\![e_1]\!] \uplus [\![e_2]\!] \qquad [\![\ominus(e)]\!] = \ominus([\![e]\!])$$

Figure 1: Typing rules and semantics for NRC$^+$.

bags can be used to simulate negation [6](see Appendix A for a discussion about the challenges posed by negation wrt. efficient maintenance within our framework).

Example 2. *Filtering an input bag R according to some predicate p can be defined in* NRC$^+$ *as:*

$$\mathrm{filter}_p[R] = \mathbf{for}\ x\ \mathbf{in}\ R\ \mathbf{where}\ p(x)\ \mathbf{union}\ \mathbf{sng}(x)$$

considering that the **for** *construct with* **where** *clause (also used in Section 2) can be expressed as follows:*

$$\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{where}\ p(x)\ \mathbf{union}\ e_2 = \mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union\ for}\ \_\ \mathbf{in}\ p(x)\ \mathbf{union}\ e_2,$$

where we ignore the variable binding the contents of the bag returned by predicate p since its only possible value is $\langle\rangle$.

For a variable X we say that an expression e is *X-dependent* if X appears as a free variable in e, and *X-independent* otherwise. Also, among NRC$^+$ expressions we distinguish between those that are *input-independent*, i.e. are *R-independent* for all relations R in the database, and those that are *input-dependent*. We define IncNRC$^+$ as the fragment of NRC$^+$ that uses a syntactically restricted singleton construct $\mathbf{sng}^*(e)$, where e must be *input-independent*. While this prevents IncNRC$^+$ queries from adding nesting levels to their inputs[5], it does provide the useful guarantee that their deltas do not require deep updates. We take advantage of this fact in the next section, when we discuss the efficient delta-processing of IncNRC$^+$. For the incrementalization of the full NRC$^+$, we provide a shredding transformation taking any NRC$^+$ query into a series of IncNRC$^+$ queries (see Section 5).

[5]We note that the query from Section 2 does not belong to IncNRC$^+$.

$$\delta_R(R) = \Delta R \qquad \delta_R(X) = \varnothing \qquad \delta_R(p(x)) = \varnothing \qquad \delta_R(\varnothing) = \varnothing$$

$$\delta_R(\textbf{let } X := e_1 \textbf{ in } e_2) = \textbf{let } X := e_1,\ \Delta X := \delta_R(e_1) \textbf{ in } \delta_R(e_2) \uplus \delta_X(e_2) \uplus \delta_R(\delta_X(e_2))$$

$$\delta_R(\textbf{sng}(x)) = \varnothing \qquad \delta_R(\textbf{sng}(\pi_i(x))) = \varnothing \qquad \delta_R(\textbf{sng}(\langle\rangle)) = \varnothing$$

$$\delta_R(\textbf{sng}^*(e)) = \varnothing \qquad \delta_R(\textbf{flatten}(e)) = \textbf{flatten}(\delta_R(e))$$

$$\delta_R(\textbf{for } x \textbf{ in } e_1 \textbf{ union } e_2) = \textbf{for } x \textbf{ in } \delta_R(e_1) \textbf{ union } e_2 \uplus \textbf{for } x \textbf{ in } e_1 \textbf{ union } \delta_R(e_2) \uplus \textbf{for } x \textbf{ in } \delta_R(e_1) \textbf{ union } \delta_R(e_2)$$

$$\delta_R(e_1 \times e_2) = \delta_R(e_1) \times e_2 \uplus e_1 \times \delta_R(e_2) \uplus \delta_R(e_1) \times \delta_R(e_2)$$

$$\delta_R(e_1 \uplus e_2) = \delta_R(e_1) \uplus \delta_R(e_2) \qquad \delta_R(\ominus(e)) = \ominus(\delta_R(e))$$

Figure 2: Delta rules for the constructs of IncNRC$^+$

4. INCREMENTALIZING IncNRC$^+$

In the following we show that any query in IncNRC$^+$ admits a delta expression with a lower cost estimate than re-evaluation. Since the derived deltas are also IncNRC$^+$ queries, their evaluation can be optimized in the same way as the original query, i.e. materialize and maintain them via delta-processing. We call the resulting expressions *higher-order* deltas. As each derivation produces 'simpler' queries, we show that the entire process has a finite number of steps and the final one is reached when the generated delta no longer depends on the database. Thus the maintenance of nested queries can be further optimized using the technique of recursive IVM, which has delivered important speedups for the flat relational case [25].

To simplify the presentation, we consider a database where a single relation R is being updated. Nonetheless, the discussion and the results carry over in a straightforward manner when updates are applied to several relations.

The delta rules for the constructs of IncNRC$^+$ wrt. the update of bag R are given in Figure 2, where ΔR is a bag containing the elements to be added/removed from R (with positive/negative multiplicity for insertions/deletions) and we use $\textbf{let } X := e_1, Y := e_2 \textbf{ in } e$ as a shorthand for $\textbf{let } X := e_1 \textbf{ in } (\textbf{let } Y := e_2 \textbf{ in } e)$. The delta of constructs that do not depend on R is the empty bag, while the rules for the other constructs are a direct consequence of their linear or distributive behavior wrt. bag union. We show that indeed, the derived delta queries $\delta_R(h)[R, \Delta R]$ produce a correct update for the return value of h :

PROPOSITION 4.1. *Given an* IncNRC$^+$ *expression* $h[R] : \textbf{Bag}(B)$ *with input* $R : \textbf{Bag}(A)$ *and update* $\Delta R : \textbf{Bag}(A)$*, then:*

$$h[R \uplus \Delta R] \;=\; h[R] \;\uplus\; \delta_R(h)[R, \Delta R].$$

PROOF. (sketch) The proof follows via structural induction on h and from the semantics of IncNRC$^+$ constructs (extended proof in [26]). □

LEMMA 1. *The delta of an* input-independent IncNRC$^+$ *expression* h *is the empty bag,* $\delta_R(h) = \varnothing$.

The lemma above is useful for deriving in a single step the delta of *input-independent* subexpressions (as in Example 3), but it also plays an important role in showing that deltas are cheaper than the original queries (Theorem 4) and in the discussion of higher-order incrementalization (Section 4.1).

Notation. We sometimes write $\delta(h)$ instead of $\delta_R(h)$ if the updated bag R can be easily inferred from the context.

EXAMPLE 3. *Taking the delta of the* IncNRC$^+$ *query presented in Example 2 results in:*

$$\delta_R(\text{filter}_p) = \textbf{for } x \textbf{ in } \Delta R \textbf{ where } p(x) \textbf{ union } \textbf{sng}(x),$$

since $\delta_R(\textbf{for } \_ \textbf{ in } p(x) \textbf{ union } \textbf{sng}(x)) = \varnothing$ *(from Lemma 1) and* $\textbf{for } x \textbf{ in } e \textbf{ union } \varnothing = \varnothing$*. As expected the delta query of* filter_p *amounts to filtering the update:* $\text{filter}_p[\Delta R]$.

4.1 Higher-order delta derivation

The technique of higher-order delta derivation stems from the intuition that if the evaluation of a query can be sped up by re-using a previous result and evaluating a cheaper delta, then the same must be true for the delta query itself. This has brought about an important leap forward in the incremental maintenance of flat queries [25], and in the following we show that our approach to delta-processing enables recursive IVM for NRC$^+$ as well (since we derive 'simpler' deltas expressed in the same language as the original query).

The delta queries $\delta(h)[R, \Delta R]$ we generate may depend on both the update ΔR as well as the initial bag R. Considering that typically the updates are much smaller than the original bags and thus the cost of evaluating $\delta(h)$ is most likely dominated by the subexpressions that depend on R, it is beneficial to partially evaluate $\delta(h)[R, \Delta R]$ offline wrt. those subexpressions that depend only on R. Once ΔR becomes available, one can use the partially evaluated expression of $\delta(h)$ to quickly compute the final update for $h[R]$.

However, since the underlying bag R is continuously being updated, in order to keep using this strategy we must be able to efficiently maintain the partial evaluation of $\delta(h)$. Fortunately, $\delta(h)[R, \Delta R]$ is an IncNRC$^+$ expression just like h, and thus we can incrementally maintain its partial evaluation wrt. R based on its second-order delta $\delta^2(h)[R, \Delta R, \Delta' R]$, as in

$$\delta(h)[R \uplus \Delta' R, \Delta R] = \delta(h)[R, \Delta R] \uplus \delta^2(h)[R, \Delta R, \Delta' R],$$

where $\Delta' R$ binds the update applied to R in $\delta(h)[R, \Delta R]$.

The same strategy can be applied to $\delta^2(h)$, leading to a series $\delta^k(h)[R, \Delta R, \cdots, \Delta^{(k-1)} R]$ of partially evaluated higher-order deltas. Each is used to incrementally maintain the preceding delta $\delta^{k-1}(h)$, all the way up to the original query h.

EXAMPLE 4. *Given bag* $R : \textbf{Bag}(\textbf{Bag}(A))$ *let us consider the first and second order deltas of query* h

$$h[R] = \textbf{flatten}(R) \times \textbf{flatten}(R)$$

$$\delta(h)[R, \Delta R] = \textbf{flatten}(R) \times \textbf{flatten}(\Delta R) \uplus \textbf{flatten}(\Delta R) \times (\textbf{flatten}(R) \uplus \textbf{flatten}(\Delta R))$$

$$\delta^2(h)[\Delta R, \Delta' R] = \textbf{flatten}(\Delta' R) \times \textbf{flatten}(\Delta R) \uplus \textbf{flatten}(\Delta R) \times \textbf{flatten}(\Delta' R).$$

In the initial stage of delta-processing, besides materializing $h[R]$ *as* H_0*, we also partially evaluate* $\delta(h)$ *wrt.* R *as* $H_1[\Delta R]$*. Then, for each update* U*, we maintain* H_0 *and* $H_1[\Delta R]$ *using:*

$$H_0 = H_0 \uplus H_1[U] \qquad H_1[\Delta R] = H_1[\Delta R] \uplus \delta^2(h)[\Delta R, U].$$

We note that one can apply updates over partially evaluated expressions like $H_1[\Delta R]$ *due to the rich algebraic structure of the calculus (bags with addition and Cartesian product form a semiring) which makes it possible to factorize*

$H_1[\Delta R] \uplus \delta^2(h)[\Delta R, U]$ *into subexpressions that depend on* ΔR*, and subexpressions that do not. Nonetheless, the process of compiling these expressions into highly optimized trigger programs is outside the scope of this work.*

Finally, we remark that in the traditional IVM approach, the value of $\mathbf{flatten}(R)$ *which depends on the entire input* R *is recomputed for each evaluation of* $\delta(h)[R, U]$*, whereas with recursive IVM we evaluate it only once during the initialization phase.*

Since we can always derive an extra delta query, this process could in principle generate an infinite series of deltas and thus render the approach of recursive IVM inapplicable. By contrast, we say that a query is *recursively incrementalizable* if there exists a k such that $\delta^k(h)$ no longer depends on the input (and therefore there is no reason to continue the recursion and to derive a delta for it). In our previous example, this happened for $k = 2$. In the following we will show that any IncNRC$^+$ query is *recursively incrementalizable*.

In order to determine the minimum k for which $\delta^k(h)$ is input-independent we associate to every IncNRC$^+$ expression a degree $\deg_\phi(h) : \mathbb{N}$ as follows: $\deg_\phi(R) = 1$, $\deg_\phi(X) = \phi(X)$, $\deg_\phi(h) = 0$ for $h \in \{\Delta R, \varnothing, p, \mathbf{sng}(x), \mathbf{sng}(\pi_i(x)), \mathbf{sng}^*(e), \mathbf{sng}(\langle\rangle)\}$ and:

$$\deg_\phi(e_1 \uplus e_2) = \max(\deg_\phi(e_1), \deg_\phi(e_2))$$
$$\deg_\phi(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2) = \deg_\phi(e_1) + \deg_\phi(e_2)$$
$$\deg_\phi(e_1 \times e_2) = \deg_\phi(e_1) + \deg_\phi(e_2)$$
$$\deg_\phi(\mathbf{flatten}(e)) = \deg_\phi(\ominus(e)) = \deg_\phi(e)$$
$$\deg_\phi(\mathbf{let}\ X := e_1\ \mathbf{in}\ e_2) = \deg_{\phi[X:=\deg_\phi(e_1)]}(e_2),$$

where ϕ associates a degree to each free variable X, corresponding to the degree of its defining expression.

We remark that the expressions h that have degree 0 are exactly those which are *input-independent*. Therefore, determining the minimum k s.t. $\delta^k(h)$ is *input-independent* means finding the minimum k s.t. $\deg(\delta^k(h)) = 0$, where $\delta^0(h) = h$. In order to show that this k is in fact the degree of h, we give the following theorem, relating the degree of an expression to the degree of its delta.

THEOREM 2. *Given an input-dependent* IncNRC$^+$ *expression* $h[R]$ *then* $\deg(\delta(h)) = \deg(h) - 1$.

PROOF. (sketch) The proof follows by induction on the structure of h and from the definition of $\delta(\cdot)$ and $\deg(\cdot)$ (extended proof in [26]). □

Theorem 2 captures the fact that the delta of a IncNRC$^+$ query is 'simpler' than the original query and we can infer from it that $\deg(\delta^k(h)) = \deg(h) - k$. It then follows that $\deg(h)$ is the minimum k s.t. $\deg(\delta^k(h))=0$, i.e. the minimum k s.t. $\delta^k(h)$ is *input-independent*.

We conclude that with recursive IVM one can avoid computing over the entire database during delta-processing by initially materializing the given query and its deltas up to $\delta^{\max(0,\deg(h)-1)}(h)$, since those are the only ones that are *input-dependent*. Then, maintaining each such materialized $H_i := \delta^i(h)$ is simply a matter of partially evaluating $\delta^{i+1}(h)$ wrt. the update and applying it to H_i. Moreover, the ability to derive higher order deltas and materialize them wrt. the database is the key result that enables the TC_0 vs. NC_0 complexity separation between nonincremental and incremental evaluation (Theorem 9).

4.2 Cost transformation

Considering that delta processing is worthwhile only if the size of the change is smaller than the original input, in this section we discuss what does it mean in the nested data model for an update to be *incremental*. Then, we provide a cost interpretation to every IncNRC$^+$ expression that given the size of its input estimates the cost of generating the output. Finally, we prove that for incremental updates the derived delta query is indeed cost-effective wrt. the original query.

While for the flat relational case incrementality can be simply defined in terms of the cardinality of the input bag wrt. the cardinality of the update, this is clearly not an appropriate measure when working with nested values, since an update of small cardinality could have arbitrarily large inner bags. In order to adequately capture and compare the size of nested values we associate to every type A of our calculus a cost domain A° equipped with a partial order and minimum values. The definition of A° is designed to preserve the distribution of cost across the nested structure of A in order to accurately reflect the size of nested values and how they impact the processing of queries operating at different nesting levels. Thus, for every type in IncNRC$^+$ we have:

$$Base^\circ = 1^\circ \qquad (A_1 \times A_2)^\circ = A_1^\circ \times A_2^\circ \qquad \mathbf{Bag}(A)^\circ = \mathbb{N}^+\{A^\circ\},$$

where 1° has only the constant cost 1, we individually track the cost of each component in a tuple, and $\mathbb{N}^+\{A^\circ\}$ represents the cost of bags as the pairing between their cardinality and the least upper-bound cost of their elements[6]. Additionally, we define a family of functions $\text{size}_A : A \to A^\circ$, that associate to any value $a : A$ a cost proportional to its size:

$$\text{size}_{Base}(x) = 1$$
$$\text{size}_{A_1 \times A_2}(\langle x_1, x_2\rangle) = \langle \text{size}_{A_1}(x_1), \text{size}_{A_2}(x_2)\rangle$$
$$\text{size}_{\mathbf{Bag}(C)}(X) = |X|\{\sup_{x_i \in X} \text{size}_C(x_i)\},$$

where the supremum function is defined based on the following type-indexed partial ordering relation $\prec_A$:

$$x \prec_{Base} y \qquad = \text{false}$$
$$\langle x_1, x_2\rangle \prec_{A_1 \times A_2} \langle y_1, y_2\rangle \qquad = x_1 \prec_{A_1} y_1 \text{ and } x_2 \prec_{A_2} y_2$$
$$n\{x\} \prec_{\mathbf{Bag}(C)} m\{y\} \qquad = n < m \text{ and } x \preceq_C y.$$

Finally, the $x \preceq_A y$ ordering is defined analogously to $\prec_A$ by making all the comparisons above non-strict, with the exception of $Base$ values for which we have $x \preceq_{Base} y = \text{true}$. We denote by 1_A the bottom element of $(A^\circ, \prec_A)$.

We can now say that an update ΔR for a nested bag R is *incremental* if $\text{size}(\Delta R) \prec \text{size}(R)$.

EXAMPLE 5. *The size of bag* R:$\mathbf{Bag}(\mathit{String} \times \mathbf{Bag}(\mathit{String}))$, $R = \{\langle \mathit{Comedy}, \{\mathit{Carnage}\}\rangle, \langle \mathit{Animation}, \{\mathit{Up}, \mathit{Shrek}, \mathit{Cars}\}\rangle\}$ *is estimated as* $\text{size}(R) : \mathbb{N}^+\{1^\circ \times \mathbb{N}^+\{1^\circ\}\} = 2\{\langle 1, 3\{1\}\rangle\}$.

Notation. Whenever the cardinality estimation of a bag is 1, we simply write $\{c\}$ as opposed to $1\{c\}$, where c is the cost estimation for its elements.

Given an IncNRC$^+$ expression $e : \mathbf{Bag}(B)$, we derive its cost $\mathcal{C}[\![e]\!] : \mathbb{N}^+\{B^\circ\}$ based on the transformation in Figure 3, where γ° and ε° are cost assignments to variables. The generated costs have two components: one that computes an upper bound for the cardinality of the output bag, denoted

[6]We use $\mathbb{N}^+\{A^\circ\}$ instead of $\mathbb{N}^+ \times A^\circ$ to distinguish it from the cost domain of tuples.

$$\mathcal{C}[\![R]\!] = \text{size}(R) \qquad \mathcal{C}[\![\mathbf{sng}(x)]\!]_{\gamma^\circ;\varepsilon^\circ} = \{\varepsilon^\circ(x)\}$$
$$\mathcal{C}[\![X]\!]_{\gamma^\circ;\varepsilon^\circ} = \gamma^\circ(X) \qquad \mathcal{C}[\![\mathbf{sng}(\pi_i(x))]\!]_{\gamma^\circ;\varepsilon^\circ} = \{\pi_i(\varepsilon^\circ(x))\}$$
$$\mathcal{C}[\![p(x)]\!] = 1_{\mathbf{Bag}(1)} \qquad \mathcal{C}[\![\mathbf{sng}(\langle\rangle)]\!] = 1_{\mathbf{Bag}(1)}$$
$$\mathcal{C}[\![\varnothing]\!] = 1_{\mathbf{Bag}(B)} \qquad \mathcal{C}[\![\mathbf{sng}^*(e)]\!] = \{\mathcal{C}[\![e]\!]\}$$
$$\mathcal{C}[\![\ominus(e)]\!] = \mathcal{C}[\![e]\!] \qquad \mathcal{C}[\![e_1 \uplus e_2]\!] = \sup(\mathcal{C}[\![e_1]\!], \mathcal{C}[\![e_2]\!])$$
$$\mathcal{C}[\![\mathbf{let}\ X := e_1\ \mathbf{in}\ e_2]\!]_{\gamma^\circ;\varepsilon^\circ} = \mathcal{C}[\![e_2]\!]_{\gamma^\circ[X:=\mathcal{C}[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ}];\varepsilon^\circ}$$
$$\mathcal{C}[\![e_1 \times e_2]\!] = \mathcal{C}_o[\![e_1]\!] \cdot \mathcal{C}_o[\![e_2]\!]\{\langle \mathcal{C}_i[\![e_1]\!], \mathcal{C}_i[\![e_2]\!]\rangle\}$$
$$\mathcal{C}[\![\mathbf{flatten}(e)]\!] = \mathcal{C}_o[\![e]\!] \cdot \mathcal{C}_{oi}[\![e]\!]\{\mathcal{C}_{ii}[\![e]\!]\}$$
$$\mathcal{C}[\![\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2]\!] =$$
$$\mathcal{C}_o[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ} \cdot \mathcal{C}_o[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[\![e_1]\!]]}\{\mathcal{C}_i[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[\![e_1]\!]]}\}$$

Figure 3: The cost transformation $\mathcal{C}[\![f]\!]{=}\mathcal{C}_o[\![f]\!]\{\mathcal{C}_i[\![f]\!]\} : \mathbb{N}^+\{B^\circ\}$ over the constructs of IncNRC$^+$.

by $\mathcal{C}_o[\![e]\!] : \mathbb{N}^+$, and another returning the upper bound for the size of its elements $\mathcal{C}_i[\![e]\!] : B^\circ$. If B is itself a bag type $\mathbf{Bag}(C)$, we also denote the two components of $\mathcal{C}_i[\![e]\!]$ by $\mathcal{C}_{oi}[\![e]\!] : \mathbb{N}^+$ and $\mathcal{C}_{ii}[\![e]\!] : C^\circ$.

The cost transformation follows the natural semantics of the constructs in IncNRC$^+$. For example, in the case of **for** x **in** e_1 **union** e_2, the cardinality of the output is estimated as the product of the cardinalities of the bags returned by e_1 and e_2, while the elements in the output have the same cost as the elements returned by e_2. We note that in computing the cost of e_2 we assigned to x the estimated cost for the elements of e_1.

Finally, we leverage the estimated cost of an expression to obtain an upper bound on its running time:

Lemma 3. *An* IncNRC$^+$ *expression* $h : \mathbf{Bag}(B)$ *can be evaluated in* $\Omega(\text{tcost}_{\mathbf{Bag}(B)}(\mathcal{C}[\![h]\!]))$*, where* $\text{tcost}_A : A^\circ \to \mathbb{N}$ *is defined as:*

$$\text{tcost}_{Base}(c) = 1 \qquad \text{tcost}_{\mathbf{Bag}(C)}(n\{c\}) = n \cdot \text{tcost}_C(c)$$
$$\text{tcost}_{A_1 \times A_2}(\langle c_1, c_2\rangle) = \text{tcost}_{A_1}(c_1) + \text{tcost}_{A_2}(c_2).$$

Proof. (Sketch) In order to show that h can be computed within $\Omega(\text{tcost}_{\mathbf{Bag}(B)}(\mathcal{C}[\![h]\!])) = \Omega(\mathcal{C}_o[\![h]\!] \cdot \text{tcost}_B(\mathcal{C}_i[\![h]\!]))$ we assume that all **let**-bound variables have been replaced by their definition and we proceed in two steps. At first we compute a lazy version of the result $h^L = [\![h]\!]^L$, which instead of inner bags produces lazy bags $\beta_{e,\varepsilon}$, i.e. closures containing the expression e that would have generated the inner bag, along with ε, the value assignment for e's free variables at the time of the evaluation. The lazy evaluation strategy $[\![\cdot]\!]^L$ operates similar to the standard interpretation $[\![\cdot]\!]$, except for the singleton construct $[\![\mathbf{sng}(e)]\!]^L_\varepsilon = \beta_{e,\varepsilon}$ and for interpreting lazy values $[\![\beta_{e,\varepsilon}]\!]^L_{\varepsilon'} = [\![e]\!]^L_\varepsilon$, for which we replace the current value assignment ε' with the one stored in the closure. Considering that producing each element of h^L takes constant time (since building tuples and closures takes constant time), it follows that this step can be done in time proportional to the cardinality of the output $O(\mathcal{C}_o[\![h]\!])$.

In the second step we expand the lazy values appearing in each element of h^L in order to obtain the final value of h. To do so we use the following expansion function:

$$\text{exp}_{Base}(x){=}x,\ \text{exp}_{A_1\times A_2}(\langle x_1, x_2\rangle){=}\langle \text{exp}_{A_1}(x_1), \text{exp}_{A_2}(x_2)\rangle$$
$$\text{exp}_{\mathbf{Bag}(C)}(\beta_{e,\varepsilon}) = \mathbf{for}\ y\ \mathbf{in}\ [\![e]\!]^L_\varepsilon\ \mathbf{union}\ \mathbf{sng}(\text{exp}_C(y)).$$

We remark that, by postponing the materialization of inner bags until after the entire top level bag has been evaluated, we avoid computing the contents of nested bags that might get projected away in a later stage of the computation (as might be the case for an eager evaluation strategy).

Our result then follows from the fact that expanding each element $x : B$ from h^L takes at most $\text{tcost}_B(\mathcal{C}_i[\![h]\!])$, which can be easily shown through induction over the structure of B and considering that $\mathcal{C}_i[\![h]\!]$ represents on upper bound for the size of the elements in the output bag. □

Example 6. *If we apply the cost transformation to the* `related`$[M]$ *query in section 2.1 we get cost estimate:*

$$\mathcal{C}[\![\texttt{related}[M]]\!] = |M|\{\langle 1, |M|\{1\}\rangle\},$$

and an upper bound for its running time as $\Omega(|M|(1+|M|))$*, which fits within the expected execution time for this query.*

We can now give the main result of this section showing that for incremental updates delta-processing is more cost-effective than recomputation.

Theorem 4. IncNRC$^+$ *is efficiently incrementalizable, i.e. for any* input-dependent IncNRC$^+$ *query* $h[R]$ *and incremental update* ΔR*, then:*

$$\text{tcost}(\mathcal{C}[\![\delta(h)]\!]) \prec \text{tcost}(\mathcal{C}[\![h]\!]).$$

Proof. (sketch) We first show by induction on the structure of h and using the cost semantics of IncNRC$^+$ constructs that $\mathcal{C}[\![\delta(h)]\!] \prec \mathcal{C}[\![h]\!]$. Then the result follows immediately from the definition of $\text{tcost}_A(\cdot)$ and $\prec_A$ (extended proof in [26]). □

It can be easily seen that $\text{filter}_p[R]$ is *efficiently incrementalizable* since its delta is $\text{filter}_p[\Delta R]$ and $\mathcal{C}[\![\text{filter}_p[R]]\!] = \mathcal{C}[\![R]\!]$, therefore $\mathcal{C}[\![\Delta R]\!] \prec \mathcal{C}[\![R]\!]$ implies $\mathcal{C}[\![\text{filter}_p[\Delta R]]\!] \prec \mathcal{C}[\![\text{filter}_p[R]]\!]$.

5. INCREMENTALIZING NRC$^+$

We now turn to the problem of efficiently incrementalizing NRC$^+$ queries that make use of the unrestricted singleton construct. As showcased in Section 2, an efficient delta rule for $\mathbf{sng}(e)$ requires deep updates which are not readily expressible in NRC$^+$. Moreover, deep updates are necessary not only for maintaining the output of a NRC$^+$ query, but also for applying local changes to the inner bags of the input. To address both problems we propose a shredding transformation that translates any NRC$^+$ query into a collection of efficiently incrementalizable expressions whose deltas can be applied via regular bag union. Furthermore, we show that our translation generates queries semantically equivalent to the original query, thus providing the first solution for the efficient delta-processing of NRC$^+$.

5.1 The shredding transformation

The essence of the shredding transformation is the replacement of inner bags by labels while separately storing their definitions in label dictionaries. Accordingly, we inductively map every type A of NRC$^+$ to a label-based/flat representation A^F along with a context component A^Γ for the corresponding label dictionaries:

$$Base^F = Base \qquad Base^\Gamma = 1$$
$$(A_1{\times}A_2)^F = A_1^F{\times}A_2^F \qquad (A_1{\times}A_2)^\Gamma = A_1^\Gamma{\times}A_2^\Gamma$$
$$\mathbf{Bag}(C)^F = \mathbb{L} \qquad \mathbf{Bag}(C)^\Gamma = (\mathbb{L} \mapsto \mathbf{Bag}(C^F)) \times C^\Gamma$$

For instance, the flat representation of a bag of type $\mathbf{Bag}(C)$ is a label $l : \mathbb{L}$, whereas its context includes a label dictionary $\mathbb{L} \mapsto \mathbf{Bag}(C^F)$, mapping l to the flattened contents of the bag.

The shredding transformation takes any NRC$^+$ expression $h[R] : \mathbf{Bag}(B)$ to:

$$\mathrm{sh}^F(h)[R^F, R^\Gamma] : \mathbf{Bag}(B^F) \quad \text{and} \quad \mathrm{sh}^\Gamma(h)[R^F, R^\Gamma] : B^\Gamma,$$

where $\mathrm{sh}^F(h)$ computes the flat representation of the output bag, while the set of queries in $\mathrm{sh}^\Gamma(h)$ define the context, i.e. the dictionaries corresponding to the labels introduced by $\mathrm{sh}^F(h)$. We note that the shredded expressions depend on the shredded input bag $R^F = \mathrm{sh}^F(R)$, $R^\Gamma = \mathrm{sh}^\Gamma(R)$[7], and that they make use of several new constructs for working with labels: the label constructor inL, the dictionary constructor $[l \mapsto e]$, and the label union of dictionaries $\cup$. We denote by NRC^+_l and IncNRC^+_l, the extension with these constructs of NRC$^+$ and IncNRC$^+$, respectively, but we postpone their formal definition until the following section. Next, we discuss some of the more interesting cases of the shredding transformation, for the full definition see Appendix B.1.

Notation. We often shorthand $\mathrm{sh}^F(h)$ and $\mathrm{sh}^\Gamma(h)$ as h^F and h^Γ, respectively. We will also abuse the notation Π/ε representing the type/value assignment for the free variables of an expression introduced by **for** constructs, to also denote a tuple type/value with one component for each such free variable.

For the unrestricted singleton construct $\mathbf{sng}(e)$ we tag each of its occurrences in an expression with a unique static index ι. Given the shredding of e, $e^F : \mathbf{Bag}(B^F)$, $e^\Gamma : B^\Gamma$, we transform $\mathbf{sng}_\iota(e)$ as follows: we first replace the inner bag e^F in its output with a label $\langle \iota, \varepsilon \rangle$ using the label constructor $\mathrm{inL}_{\iota,\Pi}$, where $\varepsilon : \Pi$ represents the value assignment for all the free variables in e^F. Since e^F operates only over shredded bags, it follows that ε is a tuple of either primitive values or labels. Then we extend the context e^Γ with a dictionary $[(\iota, \Pi) \mapsto e^F]$ mapping labels $\langle \iota, \varepsilon \rangle$ to their definition e^F:

$$\mathrm{sh}^F(\mathbf{sng}_\iota(e)) : \mathbf{Bag}(\mathbb{L}) \qquad = \mathrm{inL}_{\iota,\Pi}(\varepsilon)$$

$$\mathrm{sh}^\Gamma(\mathbf{sng}_\iota(e)) : \mathbb{L} \mapsto \mathbf{Bag}(B^F) \times B^\Gamma = \langle [(\iota, \Pi) \mapsto e^F], e^\Gamma \rangle.$$

We incorporate the value assignment ε within labels as it allows us to discuss the creation of labels independently from their defining dictionary. Also, since the value assignment ε uniquely determines the definition of a label $\langle \iota, \varepsilon \rangle$, this also ensures that we do not generate redundant label definitions. Although other alternatives can be found in the literature [9], we do not explore this issue further since our results hold independently from a particular indexing scheme.

For the shredding of $\mathbf{flatten}(e), e : \mathbf{Bag}(\mathbf{Bag}(B))$, we simply expand the labels returned by $e^F : \mathbf{Bag}(\mathbb{L})$, based on the corresponding definitions stored in the first component of the context $e^\Gamma : \mathbb{L} \mapsto \mathbf{Bag}(B^F) \times B^\Gamma$:

$$\mathrm{sh}^F(\mathbf{flatten}(e)) : \mathbf{Bag}(B^F) = \mathbf{for}\ l\ \mathbf{in}\ e^F\ \mathbf{union}\ e^{\Gamma_1}(l),$$

where $e^{\Gamma_1}/e^{\Gamma_2}$ denotes the first/second component of e^Γ.

Finally, for adding two queries in shredded form via $\uplus$, we add their flat components, but we label union their contexts, i.e. their label dictionaries:

$$\mathrm{sh}^F(e_1 \uplus e_2) = e_1^F \uplus e_2^F \qquad \mathrm{sh}^\Gamma(e_1 \uplus e_2) = e_1^\Gamma \cup e_2^\Gamma.$$

[7]We consider a full shredding of the input/output down to flat relations, although the transformation can be easily fine-tuned in order to expose only those inner bags that require updates.

To complete the shredding transformation we also inductively define $\mathrm{s}^F_A : A \to \mathbf{Bag}(A^F)$ and $\mathrm{s}^\Gamma_A : A^\Gamma$, for shredding input bags $R : \mathbf{Bag}(A)$, as well as $\mathrm{u}_A[a^\Gamma] : A^F \to \mathbf{Bag}(A)$ for converting them back to nested form, as in:

$$R^F = \mathbf{for}\ r\ \mathbf{in}\ R\ \mathbf{union}\ \mathrm{s}^F_A(r) \qquad R^\Gamma = \mathrm{s}^\Gamma_A$$

$$R = \mathbf{for}\ r^F\ \mathbf{in}\ R^F\ \mathbf{union}\ \mathrm{u}_A[R^\Gamma](r^F).$$

Shredding primitive values leaves them unchanged and produces no dictionary ($Base^\Gamma = 1$), while tuples get shredded and nested back component-wise. For shredding inner bag values we rely on an association between every bag value v in the database and a label l, as given via mappings $\mathcal{D}_C, \mathcal{D}_C^{-1}$:

$$\mathcal{D}_C : \mathbf{Bag}(C) \to \mathbf{Bag}(\mathbb{L}) \qquad \mathcal{D}_C(v) = \{l\}$$

$$\mathcal{D}_C^{-1} : \mathbb{L} \mapsto \mathbf{Bag}(C) \qquad \mathcal{D}_C^{-1}(l) = v.$$

The shredding context for these labels is then obtained by mapping each label l from the dictionary $\mathcal{D}_C^{-1}$ to a shredded version of its original value v. The full details for the definition of $\mathrm{s}^F, \mathrm{s}^\Gamma$ and u can be found in Appendix B.1.

5.2 Working with labels

In the following we detail the semantics of IncNRC^+_l's constructs for operating on dictionaries and we show that IncNRC^+_l is indeed efficiently incrementalizable.

Given an expression $e : \mathbf{Bag}(B)$ with a value assignment for its free variables $\varepsilon : \Pi$, we define a label dictionary $[(\iota, \Pi) \mapsto e] : \mathbb{L} \mapsto \mathbf{Bag}(B)$, i.e. a mapping between labels $l = \langle \iota, \varepsilon \rangle$ and bag values $e : \mathbf{Bag}(B)$, as:

$$[(\iota, \Pi) \mapsto e](\langle \iota', \varepsilon \rangle) = \text{ if } (\iota == \iota')\ \rho_\varepsilon(e) \text{ else } \{\}$$

where $\rho_\varepsilon(e)$ replaces each free variable from e with its corresponding projection from ε. A priori, such dictionaries have infinite domain, i.e. they produce a bag for each possible value assignment ε. However, when materializing them as part of a shredding context we need only compute the definitions of the labels produced by the flat version of the query.

Example 7. *Given* $\mathtt{relB}(m) : \mathbf{Bag}(String)$*, the query from the motivating example in section 2, dictionary* $d = [(\iota, Movie) \mapsto \mathtt{relB}(m)]$ *of type* $\mathbb{L} \mapsto \mathbf{Bag}(String)$ *builds a mapping between labels* $l = \langle \iota, m \rangle$ *and the bag of related movies computed by* $\mathtt{relB}(m)$*, where* l *need only range over the labels produced by* $\mathtt{related}^F$*.*

Notation. We will often abuse notation and use l to refer to both the kind of a label (ι, Π), as well as an instance of a label $\langle \iota, \varepsilon \rangle$.

In order to distinguish between an empty definition, $[] = \varnothing$, and a definition that maps its label to the empty bag, $[l \mapsto \varnothing]$, we attach support sets to label definitions such that $\mathrm{supp}([]) = \varnothing$ and $\mathrm{supp}([l \mapsto e]) = \{l\}$.

For combining dictionaries of labels, i.e. $d = [l_1 \mapsto e_1, \cdots, l_n \mapsto e_n] : \mathbb{L} \mapsto \mathbf{Bag}(B)$, with $\mathrm{supp}(d) = \{l_1, \cdots, l_n\}$, we define the addition of dictionaries $(d_1 \uplus d_2)(l) = d_1(l) \uplus d_2(l)$ as well as the *label union* of dictionaries $d_1 \cup d_2$, where $d_1, d_2 : \mathbb{L} \mapsto \mathbf{Bag}(B)$, $\mathrm{supp}(d_1 \cup d_2) = \mathrm{supp}(d_1) \cup \mathrm{supp}(d_2)$ and:

$$(d_1 \cup d_2)(l) = d_1(l), \text{ if } l \in \mathrm{supp}(d_1) \setminus \mathrm{supp}(d_2)$$

$$(d_1 \cup d_2)(l) = d_2(l), \text{ if } l \in \mathrm{supp}(d_2) \setminus \mathrm{supp}(d_1)$$

$$(d_1 \cup d_2)(l) = d_1(l), \text{ if } l \in \mathrm{supp}(d_1) \cap \mathrm{supp}(d_2)\ \&\ d_1(l) = d_2(l)$$

$$(d_1 \cup d_2)(l) = \text{error}, \text{ if } l \in \mathrm{supp}(d_1) \cap \mathrm{supp}(d_2)\ \&\ d_1(l) \neq d_2(l)$$

We ensure the well definedness of the label union operation by requiring that the definitions of labels found in both

input dictionaries must agree, i.e. for any $l \in \text{supp}(d_1) \cap \text{supp}(d_2)$ we must have $d_1(l) = d_2(l)$. If this condition is not met the evaluation of $\cup$ will result in an error. We remark that $\cup$ cannot modify a label definition, only $\uplus$ can (for an example contrasting their semantics see Appendix B.2). Moreover, we formalize the notion of consistent shredded values, i.e. values that do not contain undefined labels or definitions that conflict and we show that shredding produces consistent values and that given consistent inputs, shredded NRC^+ expressions also produce consistent outputs [26].

Finally, we introduce the delta rules and the degree and cost interpretations for the new label-related constructs:

$$\delta([l \mapsto e]) = [l \mapsto \delta(e)] \quad \delta(\text{inL}_l) = \varnothing \quad \delta(e_1 \cup e_2) = \delta(e_1) \cup \delta(e_2)$$
$$\deg([l \mapsto e]) = \deg(e) \qquad \deg(\text{inL}_l) = 0$$
$$\deg(e_1 \cup e_2) = \max(\deg(e_1), \deg(e_2))$$
$$\mathcal{C}[\![[l \mapsto e](l')]\!] = \mathcal{C}[\![e]\!] \qquad \mathcal{C}[\![\text{inL}_l(a)]\!] = \{1\}$$
$$\mathcal{C}[\![(e_1 \cup e_2)(l)]\!] = \sup(\mathcal{C}[\![e_1(l)]\!], \mathcal{C}[\![e_2(l)]\!]),$$

where the cost domains for labels is 1°. Based on these definitions we prove the following result:

THEOREM 5. *IncNRC^+_l is recursively and efficiently incrementalizable, i.e. given any input-dependent IncNRC^+_l query $h[R]$, and incremental update ΔR then:*

$$h[R \uplus \Delta R] = h[R] \uplus \delta(h)[R, \Delta R], \quad \deg(\delta(h)) = \deg(h) - 1$$
$$\text{and} \qquad \text{tcost}(\mathcal{C}[\![\delta(h)]\!]) < \text{tcost}(\mathcal{C}[\![h]\!]).$$

Theorem 5 implies that we can efficiently incrementalize any NRC^+ query by incrementalizing the IncNRC^+_l queries resulting from its shredding. The output of these queries faithfully represents the expected nested value as we demonstrate in section 5.3.

5.3 Correctness

In order to prove the correctness of the shredding transformation, we show that for any NRC^+ query $h[R] : \mathbf{Bag}(B)$, shredding the input bag $R : \mathbf{Bag}(A)$, evaluating h^F, h^Γ, and converting the output back to nested form produces the same result as $h[R]$, that is:

$$h[R] = \textbf{let } R^F := \textbf{for } r \textbf{ in } R \textbf{ union } \text{s}^F(r),\ R^\Gamma := \text{s}^\Gamma \textbf{ in}$$
$$\textbf{for } x^F \textbf{ in } h^F \textbf{ union } \text{u}[h^\Gamma](x^F), \tag{4}$$

where $\text{s}^F(r)$ shreds each tuple in R to its flat representation, s^Γ returns the dictionaries corresponding to the labels generated by $\text{s}^F(r)$, and $\text{u}[h^\Gamma](x^F)$ places each tuple from h^F back in nested form using the dictionaries in h^Γ.

We proceed with the proof in two steps. We first show that shredding a value and then nesting the result returns back the original value (Lemma 6). Then, we show that applying the shredded version of a function over a shredded value and then nesting the result is equivalent to first nesting the input and then applying the original function (Lemma 7). The main result then follows immediately (Theorem 8). The proofs of the following lemmas rely on induction over the structure of the types or expressions involved, and are omitted for space reasons (they can be found in [26]).

LEMMA 6. *The nesting function* u *is left inverse wrt. the shredding functions* $\text{s}^F, \text{s}^\Gamma$*, i.e. for nested value* $a : A$ *we have* $\textbf{for } a^F \textbf{ in } \text{s}^F_A(a) \textbf{ union } \text{u}_A[\text{s}^\Gamma_A](a^F) = \textbf{sng}(a)$.

LEMMA 7. *For any NRC^+ query $h[R] : \mathbf{Bag}(B)$ and consistent shredded bag R^F, R^Γ:*

$$\textbf{let } R := (\textbf{for } r^F \textbf{ in } R^F \textbf{ union } \text{u}[R^\Gamma](r^F)) \textbf{ in } h[R]$$
$$= \textbf{for } x^F \textbf{ in } h^F \textbf{ union } \text{u}[h^\Gamma](x^F).$$

THEOREM 8. *For any NRC^+ query property (4) holds.*

PROOF. The result follows from Lemma 7, if we consider the shredding of R as input, and then apply Lemma 6. □

5.4 Complexity class separation

In terms of data complexity, NRC belongs to TC_0 [39, 23], the class of languages recognizable by LOGSPACE-uniform families of circuits of polynomial size and constant depth using and-, or- and majority-gates of unbounded fan-in. The positive fragment of NRC is in the same complexity class since just the flatten operation on bag semantics requires the power to compute the sum of integers, which is in TC_0. In the following, we show that incrementalizing NRC^+ queries in shredded form fits within the strictly lower complexity class of NC_0, which is a better model for real hardware since, in contrast to TC_0, it uses only gates with bounded fan-in. To obtain this result we require that multiplicities are represented by fixed size integers of k bits, and thus their value is computed modulo 2^k.

Assume that, for the following circuit complexity proof, shredded values are available as a bit sequence, with k bits (representing a multiplicity modulo 2^k) for each possible tuple constructible from the active domain of the shredded views and their schema, in some canonical ordering. For $k = 1$, this is the standard representation for circuit complexity proofs for relational queries with set semantics. Note that the active domain of a shredded view consists of the active domain of the nested value it is constructed from, the delimiters "⟨", "⟩", ",", "{", "}", as well as an additional linearly-sized label set. We consider this the *natural* bit sequence representation of shredded values.

It may be worth pointing out that shredding only creates polynomial blow-up compared to a string representation of a complex value (e.g. in XML or JSON). This further justifies our representation. Generalizing the classical bit representation of relational databases (which has polynomial blow-up) to non-first normal form relations (with, for the simplest possible type $\{\langle\{Base\}\rangle\}$, one bit for every possible subset of the active domain) has exponential blow-up.

THEOREM 9. *Materialized views of NRC^+ queries with multiplicities modulo 2^k in shredded form are incrementally maintainable in NC_0 wrt. constant size updates.*

PROOF. We will refer to the database and the update by d and Δd, respectively. By Theorem 8, every NRC^+ query can be simulated by a fixed number of IncNRC^+ queries on the shredding of the input. By Proposition 4.1, for every IncNRC^+ query h, there is an IncNRC^+ query $\delta_d(h)$ such that $h(d \uplus \Delta d) = h(d) \uplus \delta_d(h)(d)(\Delta d)$. We partially evaluate and materialize such delta queries as views $h' := \delta_d(h)(d)$ which then allow lookup of $h'(\Delta d)$. By Theorem 2, given an IncNRC^+ query h, there is a a finite stack of higher-order delta queries $h_0, \cdots, h_k$ (with $h_i = \delta^{(i)}_d(h)(d)$, $0 \le i \le k$, and $\delta^{(0)}_d(h)(d) = h(d)$) such that h_k is input-independent (only depends on Δd). Thus, h_i can be refreshed as $h_i := h_i \uplus h_{i+1}(\Delta d)$ for $i < k$. We can incrementally maintain overall query h on a group of views in shredded representation using just the $\uplus$ operations and the operations of IncNRC^+ on a

constant-size input (executing queries h_i on the update). This is all the work that needs to be done, for an update, to refresh all the views.

It is easy to verify that in natural bit sequence representation of the shredded views, both $\uplus$ (on the full input representations) and IncNRC$^+$ on constantly many input bits can be modeled using NC_0 circuit families, one for each meaningful size of input bit sequence. For IncNRC$^+$ on constant-size inputs, this is obvious, since all Boolean functions over constantly many input bits can the captured by constant-size bounded fan-in circuits, and since there is really only one circuit, it can also be output in LOGSPACE. For $\uplus$, remember that we represent multiplicities modulo 2^k, i.e. by a fixed k bits. Since addition modulo 2^k is in NC_0, so is $\uplus$: The view contains aggregate multiplicities, each of which only needs to be combined with one multiplicity from the respective delta view. The overall circuit for an input size is a straightforward composition of these building blocks. □

In contrast, even when multiplicities are modeled modulo 2^k and the input is presented in flattened form, NRC$^+$ is not in NC_0 since multiplicities of projections (or **flatten**) depend on an unbounded number of input bits.

In Appendix B.3, we show that shredding (for the initial materialization of the views) itself is in TC_0; it follows immediately that shredding constant-size updates – the only shredding necessary during IVM – is in NC_0.

6. RELATED WORK

Delta derivation was originally proposed for datalog programs [19, 20] but it is even more natural for algebraic query languages such as the relational algebra on bags [16, 5, 8, 37, 40], simply because the algebraic structure of a group is the necessary and sufficient environment in which deltas live. In many cases the derived deltas are asymptotically faster than the original queries and the resulting speedups prompted a wide adoption of such techniques in commercial database systems. Our work is an attempt to develop similarly powerful static incrementalization tools for languages on nested collections and comes in the context of advances in the complexity class separation between recomputation and IVM [24, 42]. Compared to [24] which discusses the recursive incrementalization of a flat query language, we address the challenges raised by a nested data model, i.e. we design a *closed* delta transformation for IncNRC$^+$'s constructs and a semantics-preserving shredding transformation for implementing 'deep' updates. Furthermore, we provide cost domains and a cost interpretations for IncNRC$^+$'s constructs, according to which we define the notion of an *incremental* nested update and we show that the deltas we generate have lower upper-bound time estimates than re-evaluation.

The **nested data model** has been thoroughly studied in the literature over multiple decades and has enjoyed a wide adoption in industry in the form of data format standards like XML or JSON. However, solutions to the problem of incremental maintenance for nested queries either focus only on the fragment of the language that does not generate changes to inner collections [15], or propagate those changes based on auxiliary data structures designed to track the lineage of tuples in the view [14, 13, 22, 34]. The use of dedicated data-structures as well as custom update languages make it extremely difficult to further apply recursive IVM on top of these techniques. In contrast, our approach is fully algebraic and both the given query as well as the generated deltas belong to the same language and thus they can be further incrementalized via higher-order delta processing.

The related topic of **incremental computation** has also received considerable attention within the programming languages community, with proposals being divided between dynamic and static approaches. The dynamic solutions, such as *self-adjusting computation* [3, 1, 2], record at runtime the dependency-graph of the computation. Then, upon updates, one can easily identify the intermediate results affected and trigger their re-evaluation. As this technique makes few assumptions about its target language, it is applicable to a variety of languages ranging from Standard ML to C. Nonetheless, its generality comes at the price of significant runtime overheads for building the dependency graph. Moreover, while static solutions derive deltas that can be further optimized via global transformations, such an opportunity is mostly missed by dynamic approaches .

Delta derivation has also been proposed in the context of incremental computation, initially only for first-order languages [35], and more recently it has been extended to higher-order languages [7]. However, these approaches offer no guarantees wrt. the efficiency of the generated deltas, whereas in our work we introduce cost interpretations and discuss the requirements for cost-efficient delta processing.

The challenge of **shredding** nested queries has been previously addressed by Paredaens et al. [36], who propose a translation taking flat-to-flat nested relational algebra expressions into flat relational algebra. Van den Bussche [10] also showed that it is possible to evaluate nested queries over sets via multiple flat queries, but his solution may produce results that are quadratically larger than needed [9].

Shredding transformations have been studied more recently in the context of language integrated querying systems such as Links [31] and Ferry [17]. In order to efficiently evaluate a nested query, it is first converted to a series of flat queries which are then sent to the database engine for execution. While these transformations also replace inner collections with flat values, they are geared towards generating SQL queries and thus they make assumptions that are not applicable to our goal of efficiently incrementalizing any nested-to-nested expressions. For example, Ferry makes extensive use of On-Line Analytic Processing (OLAP) features of SQL:1999, such as `ROW_NUMBER` and `DENSE_RANK` [18], while Links relies on a normalization phase and handles only flat-to-nested expressions [9]. More importantly, none of the existing proposals translate NRC$^+$ queries to an efficiently incrementalizable language.

7. REFERENCES

[1] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proc. POPL*, pages 309–322, 2008.

[2] Umut A. Acar, Guy Blelloch, Ruy Ley-Wild, Kanat Tangwongsan, and Duru Turkoglu. Traceable data types for self-adjusting computation. In *Proc. PLDI*, pages 483–496, 2010.

[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proc. POPL*, pages 247–259, 2002.

[4] David A. M. Barrington, Neil Immerman, and H. Straubing. "On Uniformity within NC1". *Journal of Computer and System Sciences*, **41**(3):274–306, 1990.

[5] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In

Proc. SIGMOD Conference, pages 61–71, 1986.

[6] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.

[7] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *Proc. PLDI*, pages 145–155, 2014.

[8] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.

[9] James Cheney, Sam Lindley, and Philip Wadler. Query shredding: Efficient relational evaluation of queries over nested multisets. In *Proc. SIGMOD*, pages 1027–1038, 2014.

[10] Jan Van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science*, 254(1–2):363 – 377, 2001.

[11] Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. Well-definedness and semantic type-checking for the nested relational calculus. *Theor. Comput. Sci.*, 371(3):183–199, 2007.

[12] Jan Van den Bussche and Stijn Vansummeren. Well-defined NRC queries can be typed - (extended abstract). In *In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman*, pages 494–506, 2013.

[13] Katica Dimitrova, Maged El-Sayed, and ElkeA. Rundensteiner. Order-sensitive view maintenance of materialized xquery views. In *Conceptual Modeling - ER 2003*, volume 2813 of *Lecture Notes in Computer Science*, pages 144–157. 2003.

[14] J. Nathan Foster, Ravi Konuru, Jérôme Siméon, and Lionel Villard. An algebraic approach to view maintenance for XQuery. In *PLAN-X 2008, Programming Language Technologies for XML*.

[15] Dieter Gluche, Torsten Grust, Christof Mainberger, and MarcH. Scholl. Incremental updates for materialized oql views. In *Deductive and Object-Oriented Databases*, volume 1341 of *Lecture Notes in Computer Science*, pages 52–66. 1997.

[16] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *Proc. SIGMOD*, pages 328–339, 1995.

[17] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: Database-supported program execution. In *Proc. Conference on Management of Data*, SIGMOD '09, pages 1063–1066, 2009.

[18] Torsten Grust, Jan Rittinger, and Tom Schreiber. Avalanche-safe linq compilation. *Proc. VLDB Endow.*, 3(1-2):162–172, 2010.

[19] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. Counting solutions to the view maintenance problem. In *Proc. Workshop on Deductive Databases, JICSLP*, 1992.

[20] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD'93*, pages 157–166.

[21] David S. Johnson. A catalog of complexity classes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 1, chapter 2, pages 67–161. Elsevier Science Publishers B.V., 1990.

[22] Akira Kawaguchi, Daniel Lieuwen, Inderpal Mumick, and Kenneth Ross. Implementing incremental view maintenance in nested data models. In *In Proceedings of the Workshop on Database Programming Languages*, pages 202–221, 1997.

[23] Christoph Koch. On the complexity of nonrecursive xquery and functional query languages on complex values. In *Proc. PODS*, pages 84–97, 2005.

[24] Christoph Koch. Incremental query evaluation in a ring of databases. In *Proc. PODS*, pages 87–98, 2010.

[25] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.

[26] Christoph Koch, Daniel Lupei, and Val Tannen. Incremental view maintenance for collection programming. *CoRR*, abs/1412.4320, 2016.

[27] S. Kazem Lellahi and Val Tannen. A calculus for collections and aggregates. In *Category Theory and Computer Science, 7th International Conference, CTCS '97, Proceedings*, pages 261–280.

[28] Alon Y. Levy and Dan Suciu. Deciding containment for queries with complex objects. In *Proc. PODS*, pages 20–31, 1997.

[29] Leonid Libkin and Limsoon Wong. Query languages for bags and aggregate functions. *J. Comput. Syst. Sci.*, 55(2):241–272, 1997.

[30] Hartmut Liefke and Susan B. Davidson. Specifying updates in biomedical databases. In *Proc. SSDBM*, 1999.

[31] Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proc. Workshop on Types in Language Design and Implementation*, TLDI '12, pages 91–102, 2012.

[32] Jixue Liu, Millist W. Vincent, and Mukesh K. Mohania. Incremental evaluation of nest and unnest operators in nested relations. In *Proc. of 1999 CODAS Conf*, pages 264–275, 1999.

[33] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: Reconciling object, relations and xml in the .net framework. In *Proc. SIGMOD*, pages 706–706, 2006.

[34] Hiroaki Nakamura. Incremental computation of complex object queries. OOPSLA, pages 156–165, 2001.

[35] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, 1982.

[36] Jan Paredaens and Dirk Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Trans. Database Syst.*, 17(1):65–93, March 1992.

[37] Nick Roussopoulos. An incremental access method for viewcache: Concept, algorithms, and cost analysis. *ACM Transactions on Database Systems*, 16(3):535–563, 1991.

[38] Dan Suciu. Bounded fixpoints for complex objects. In *Database Programming Languages (DBPL-4), Proc. of the Fourth International Workshop on Database Programming Languages - Object Models and Languages*, pages 263–281, 1993.

[39] Dan Suciu and Val Tannen. "A Query Language for NC". In *Proc. PODS'94*, pages 167–178, 1994.

[40] Dimitra Vista. Integration of incremental view maintenance into query optimizers. In *Advances in Database Technology-EDBT'98*, volume 1377, pages 374–388, 1998.

[41] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. HotCloud'10, 2010.

[42] Thomas Zeume and Thomas Schwentick. Dynamic conjunctive queries. In *Proc. ICDT*, pages 38–49, 2014.

APPENDIX

A. CHALLENGES FOR EFFICIENT INCREMENTALIZATION

In the following we discuss the challenges in deriving a delta query which is cheaper than full re-evaluation for any expression in a language.

Informally, we say that the delta $\delta(e)[R, \Delta R]$ of a query $e[R]$ is more *efficient* than full recomputation (or simply *efficient*), if for any update ΔR s.t. $\text{size}(\Delta R) \ll \text{size}(R)$, evaluating $\delta(e)[R, \Delta R]$ and applying it to the output of e is less expensive than re-evaluating e from scratch, i.e.:

$$\text{cost}(\delta(e)[R, \Delta R]) \ll \text{cost}(e[R \uplus \Delta R]) \quad \text{and}$$
$$\text{size}(\delta(e)[R, \Delta R]) \ll \text{size}(e[R \uplus \Delta R]),$$

where the second equation ensures that applying the update is cheaper than re-computation considering that the cost of applying an update is proportional to its size and that the cost of evaluating an expression is lowerbounded by the size of its output ($\text{size}(e[R \uplus \Delta R]) \leq \text{cost}(e[R \uplus \Delta R])$).

One can guarantee that the delta of any expression in a language is efficient by requiring that every construct $\mathbf{p}(e)[R]$ of the language satisfies the property above, i.e. $\text{size}(\delta(e)[R, \Delta R]) \ll \text{size}(e[R])$ implies:

$$\text{cost}(\delta(\mathbf{p}(e))[R, \Delta R]) \ll \text{cost}(\mathbf{p}(e)[R \uplus \Delta R]) \quad \text{and}$$
$$\text{size}(\delta(\mathbf{p}(e))[R, \Delta R]) \ll \text{size}(\mathbf{p}(e)[R \uplus \Delta R]) \tag{5}$$

Unfortunately, this property does not hold for constructs $\mathbf{p}(e)[R]$ which take linear time in their inputs $e[R]$ (i.e. $\text{cost}(\mathbf{p}(e)[R]) = \text{size}(e[R])$) and whose delta $\delta(\mathbf{p}(e))[R, \Delta R]$ depends on the original input $e[R]$ (therefore $\text{cost}(e[R]) < \text{cost}(\delta(\mathbf{p}(e))[R, \Delta R])$), as it leads to the following contradiction:

$$\text{size}(e[R]) \leq \text{cost}(e[R]) < \text{cost}(\delta(\mathbf{p}(e))[R, \Delta R]) \ll$$
$$\ll \text{cost}(\mathbf{p}(e)[R \uplus \Delta R]) = \text{size}(e[R \uplus \Delta R]) \approx \text{size}(e[R]),$$

where the last approximation follows from the fact that:

$$e[R \uplus \Delta R] = e[R] \ \uplus \ \delta(e)[R, \Delta R] \quad \text{and}$$
$$\text{size}(\delta(e)[R, \Delta R]) \ll \text{size}(e[R]).$$

An example of such a construct is bag subtraction $(e_1 \setminus e_2)[R]$, that associates to every element v_i in $e_1[R]$ the multiplicity $\max(0, m_1 - m_2)$, where m_1, m_2 are v_i's multiplicities in $e_1[R]$ and $e_2[R]$, respectively. Indeed, the cost of evaluating bag subtraction is proportional to its input (i.e. $\text{cost}(e_1 \setminus e_2)[R] = \text{size}(e_1[R])$, assuming $e_1[R]$ and $e_2[R]$ have similar sizes) and the result of $(e_1 \setminus e_2)[R]$ can be maintained when $e_2[R]$ changes, only if the initial value of $e_1[R]$ is known at the time of the update. The singleton constructor or the emptiness test over bags also exhibit similar characteristics. By contrast, constructs that take time linear in their input, but whose delta rule depends only on the update do not present this issue (eg. **flatten**).

This problem can be addressed by materializing the result of the subquery $e[R]$, such that one does not need to pay its cost again when evaluating $\delta(\mathbf{p}(e))[R, \Delta R]$. However, this only solves half of the problem, as we also need to make sure that the outcome of $\delta(\mathbf{p}(e))[R, \Delta R]$ can be efficiently propagated through outer queries e' that may use $\mathbf{p}(e)[R \uplus \Delta R]$ as a subquery. Solving this issue requires handcrafted solutions that take into consideration the particularities of $\mathbf{p}$ and the ways it can be used. For example, in our solution for efficiently incrementalizing $\mathbf{sng}(\cdot)$ we take advantage of the fact that the only way of accessing the contents of a inner bag is via $\mathbf{flatten}(\cdot)$.

Finally, for constructs $\mathbf{p}$ with boolean as output domain (eg. testing whether a bag is empty), it no longer makes sense to distinguish between small and large values, and therefore, the condition (5) can never be satisfied. This problem extends to a class of primitives that includes bag equality, negation, and membership testing, and restricts our solution for efficient incrementalization to only the positive fragment of nested relational calculus NRC$^+$.

B. SHREDDING NRC$^+$

B.1 The shredding transformation

The full definition of the shredding transformation for the constructs of NRC$^+$ can be found in Figure 4. We remark that it produces expressions that no longer make use of the singleton combinator $\mathbf{sng}(e)$, thus their deltas do not generate any deep updates.

In addition, we note that only the shreddings of $\mathbf{sng}(e)$ and $\mathbf{flatten}(e)$ fundamentally change the contexts, whereas the shreddings of most of the other operators modify only the flat component of the output (see $\text{sh}(e_1 \times e_2)$, $\text{sh}(\ominus(e))$). In fact, if we interpret the output context B^Γ as a tree, having the same structure as the nested type B, we can see that $\text{sh}^\Gamma(\mathbf{sng}(e))$ / $\text{sh}^\Gamma(\mathbf{flatten}(e))$ are the only ones able to add / remove a level from the tree.

We define $\text{s}^F_A : A \to \mathbf{Bag}(A^F)$ and $\text{s}^\Gamma_A : A^\Gamma$, for shredding bag values $R : \mathbf{Bag}(A)$, as well as $\text{u}_A[a^\Gamma] : A^F \to \mathbf{Bag}(A)$ for converting them back to nested form:

$$R^F = \mathbf{for}\ a\ \mathbf{in}\ R\ \mathbf{union}\ \text{s}^F_A(a) \qquad R^\Gamma = \text{s}^\Gamma_A \qquad\qquad R = \mathbf{for}\ a^F\ \mathbf{in}\ R^F\ \mathbf{union}\ \text{u}_A[R^\Gamma](a^F),$$

where $\text{s}^F_A, \text{s}^\Gamma_A$ and u_A are presented in Figure 5.

$\mathrm{sh}^F(\mathbf{sng}(x)) : \mathbf{Bag}(A^F)$ $\quad \mathrm{sh}^F(\mathbf{sng}(x)) = \mathbf{sng}(x^F)$ $\quad \mathrm{sh}^\Gamma(\mathbf{sng}(x)) : A^\Gamma$ $\quad \mathrm{sh}^\Gamma(\mathbf{sng}(x)) = x^\Gamma$

$\mathrm{sh}^F(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2) : \mathbf{Bag}(B^F)$ $\quad \mathrm{sh}^F(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2) = \mathbf{let}\ x^\Gamma := e_1^\Gamma\ \mathbf{in\ for}\ x^F\ \mathbf{in}\ e_1^F\ \mathbf{union}\ e_2^F$ $\quad \mathrm{sh}^\Gamma(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2) : B^\Gamma$ $\quad \mathrm{sh}^\Gamma(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2) = \mathbf{let}\ x^\Gamma := e_1^\Gamma\ \mathbf{in}\ e_2^\Gamma$

$\mathrm{sh}^F(\mathbf{sng}(\pi_i(x))) : \mathbf{Bag}(A_i^F)$ $\quad \mathrm{sh}^F(\mathbf{sng}(\pi_i(x))) = \mathbf{sng}(\pi_i(x^F))$ $\quad \mathrm{sh}^\Gamma(\mathbf{sng}(\pi_i(x))) : A_i^\Gamma$ $\quad \mathrm{sh}^\Gamma(\mathbf{sng}(\pi_i(x))) = x^{\Gamma_i}$

$\mathrm{sh}^F(\mathbf{sng}(\langle\rangle)) : \mathbf{Bag}(1)$ $\quad \mathrm{sh}^F(\mathbf{sng}(\langle\rangle)) = \mathbf{sng}(\langle\rangle)$ $\quad \mathrm{sh}^\Gamma(\mathbf{sng}(\langle\rangle)) : 1$ $\quad \mathrm{sh}^\Gamma(\mathbf{sng}(\langle\rangle)) = \langle\rangle$

$\mathrm{sh}^F(e_1 \times e_2) : \mathbf{Bag}(A_1^F \times A_2^F)$ $\quad \mathrm{sh}^F(e_1 \times e_2) = e_1^F \times e_2^F$ $\quad \mathrm{sh}^\Gamma(e_1 \times e_2) : A_1^\Gamma \times A_2^\Gamma$ $\quad \mathrm{sh}^\Gamma(e_1 \times e_2) = \langle e_1^\Gamma, e_2^\Gamma \rangle$

$\mathrm{sh}^F(R) : \mathbf{Bag}(A^F)$ $\quad \mathrm{sh}^F(R) = \mathbf{for}\ r\ \mathbf{in}\ R\ \mathbf{union}\ \mathrm{s}_A^F(r)$ $\quad \mathrm{sh}^\Gamma(R) : A^\Gamma$ $\quad \mathrm{sh}^\Gamma(R) = \mathrm{s}_A^\Gamma$

$\mathrm{sh}^F(e_1 \uplus e_2) : \mathbf{Bag}(B^F)$ $\quad \mathrm{sh}^F(e_1 \uplus e_2) = e_1^F \uplus e_2^F$ $\quad \mathrm{sh}^\Gamma(e_1 \uplus e_2) : B^\Gamma$ $\quad \mathrm{sh}^\Gamma(e_1 \uplus e_2) = e_1^\Gamma \cup e_2^\Gamma$

$\mathrm{sh}^F(\mathbf{sng}_\iota(e)) : \mathbf{Bag}(\mathbb{L})$ $\quad \mathrm{sh}^F(\mathbf{sng}_\iota(e)) = \mathrm{inL}_{\iota,\Pi}(\varepsilon)$ $\quad \mathrm{sh}^\Gamma(\mathbf{sng}_\iota(e)) : (\mathbb{L} \to \mathbf{Bag}(B^F)) \times B^\Gamma$ $\quad \mathrm{sh}^\Gamma(\mathbf{sng}_\iota(e)) = \langle [(\iota, \Pi) \mapsto e^F], e^\Gamma \rangle$

$\mathrm{sh}^F(\varnothing) : \mathbf{Bag}(B^F)$ $\quad \mathrm{sh}^F(\varnothing) = \varnothing$ $\quad \mathrm{sh}^\Gamma(\varnothing) : B^\Gamma$ $\quad \mathrm{sh}^\Gamma(\varnothing) = \varnothing_{B^\Gamma}$

$\mathrm{sh}^F(\ominus(e)) : \mathbf{Bag}(B^F)$ $\quad \mathrm{sh}^F(\ominus(e)) = \ominus(e^F)$ $\quad \mathrm{sh}^\Gamma(\ominus(e)) : B^\Gamma$ $\quad \mathrm{sh}^\Gamma(\ominus(e)) = e^\Gamma$

$\mathrm{sh}^F(\mathbf{flatten}(e)) : \mathbf{Bag}(B^F)$ $\quad \mathrm{sh}^F(\mathbf{flatten}(e)) = \mathbf{for}\ l\ \mathbf{in}\ e^F\ \mathbf{union}\ e^{\Gamma_1}(l)$ $\quad \mathrm{sh}^\Gamma(\mathbf{flatten}(e)) : B^\Gamma$ $\quad \mathrm{sh}^\Gamma(\mathbf{flatten}(e)) = e^{\Gamma_2}$

$\mathrm{sh}^F(p(x)) : \mathbf{Bag}(1)$ $\quad \mathrm{sh}^F(p(x)) = p(x)$ $\quad \mathrm{sh}^\Gamma(p(x)) : 1$ $\quad \mathrm{sh}^\Gamma(p(x)) = \langle\rangle$

$\mathrm{sh}^F(\mathbf{let}\ X := e_1\ \mathbf{in} e_2) : \mathbf{Bag}(B^F)$ $\quad \mathrm{sh}^F(\mathbf{let}\ X := e_1\ \mathbf{in}\ e_2) = \mathbf{let}\ X^F := e_1^F,\ X^\Gamma := e_2^\Gamma\ \mathbf{in}\ e_2^F$
$\mathrm{sh}^\Gamma(\mathbf{let}\ X := e_1\ \mathbf{in} e_2) : B^\Gamma$ $\quad \mathrm{sh}^\Gamma(\mathbf{let}\ X := e_1\ \mathbf{in}\ e_2) = \mathbf{let}\ X^F := e_1^F,\ X^\Gamma := e_2^\Gamma\ \mathbf{in}\ e_2^\Gamma$

Figure 4: The shredding transformation, where s_A^F and s_A^Γ are described in Figure 5a.

$\mathrm{s}^F_{Base} : Base \to \mathbf{Bag}(Base)$ $\quad \mathrm{s}^F_{Base}(a) = \mathbf{sng}(a)$

$\mathrm{s}^F_{A_1 \times A_2} : (A_1 \times A_2) \to \mathbf{Bag}(A_1^F \times A_2^F)$ $\quad \mathrm{s}^F_{A_1 \times A_2}(a) = \mathbf{for}\ \langle a_1, a_2 \rangle\ \mathbf{in}\ \mathbf{sng}(a)\ \mathbf{union}\ \mathrm{s}^F_{A_1}(a_1) \times \mathrm{s}^F_{A_2}(a_2)$

$\mathrm{s}^F_{\mathbf{Bag}(C)} : \mathbf{Bag}(C) \to \mathbf{Bag}(\mathbb{L})$ $\quad \mathrm{s}^F_{\mathbf{Bag}(C)}(v) = \mathcal{D}_C(v)$

$\mathrm{s}^\Gamma_{Base} : 1$ $\quad \mathrm{s}^\Gamma_{Base} = \langle\rangle$

$\mathrm{s}^\Gamma_{A_1 \times A_2} : A_1^\Gamma \times A_2^\Gamma$ $\quad \mathrm{s}^\Gamma_{A_1 \times A_2} = \langle \mathrm{s}^\Gamma_{A_1}, \mathrm{s}^\Gamma_{A_2} \rangle$

$\mathrm{s}^\Gamma_{\mathbf{Bag}(C)} : (\mathbb{L} \mapsto \mathbf{Bag}(C^F)) \times C^\Gamma$ $\quad \mathrm{s}^{\Gamma_1}_{\mathbf{Bag}(C)} = \mathbf{for}\ l\ \mathbf{in}\ \mathrm{supp}(\mathcal{D}_C^{-1})\ \mathbf{union}\ [l \mapsto \mathbf{for}\ c\ \mathbf{in}\ \mathcal{D}_C^{-1}(l)\ \mathbf{union}\ \mathrm{s}_C^F(c)]$ $\quad \mathrm{s}^{\Gamma_2}_{\mathbf{Bag}(C)} = \mathrm{s}_C^\Gamma$

(a) $\mathrm{s}_A^F : A \to \mathbf{Bag}(A^F), \mathrm{s}_A^\Gamma : A^\Gamma$

$\mathrm{u}_{Base}[\langle\rangle] : Base \to \mathbf{Bag}(Base)$ $\quad \mathrm{u}_{Base}[\langle\rangle](a^F) = \mathbf{sng}(a^F)$

$\mathrm{u}_{A_1 \times A_2}[a^\Gamma] : A_1^F \times A_2^F \to \mathbf{Bag}(A_1 \times A_2)$ $\quad \mathrm{u}_{A_1 \times A_2}[a^\Gamma](a^F) = \mathbf{for}\ \langle a_1^F, a_2^F \rangle\ \mathbf{in}\ \mathbf{sng}(a^F)\ \mathbf{union}\ \mathrm{u}_{A_1}[a^{\Gamma_1}](a_1^F) \times \mathrm{u}_{A_2}[a^{\Gamma_2}](a_2^F)$

$\mathrm{u}_{\mathbf{Bag}(C)}[a^\Gamma] : \mathbb{L} \to \mathbf{Bag}(\mathbf{Bag}(C))$ $\quad \mathrm{u}_{\mathbf{Bag}(C)}[a^\Gamma](l) = \mathbf{sng}(\mathbf{for}\ c^F\ \mathbf{in}\ a^{\Gamma_1}(l)\ \mathbf{union}\ \mathrm{u}_C[a^{\Gamma_2}](c^F))$

(b) $\mathrm{u}_A[a^\Gamma] : A^F \to \mathbf{Bag}(A)$

Figure 5: Shredding and nesting bag values.

When shredding a bag value $R : \mathbf{Bag}(A)$, the flat component $R^F : \mathbf{Bag}(A^F)$ is generated by replacing every nested bag $v : \mathbf{Bag}(C)$ from R, with a label $l = \langle \iota_v, \langle\rangle \rangle$. The association between every bag $v : \mathbf{Bag}(C)$, occurring nested somewhere inside R, and the label l is given via $\mathcal{D}_C$ and $\mathcal{D}_C^{-1}$:

$$\mathcal{D}_C : \mathbf{Bag}(C) \to \mathbf{Bag}(\mathbb{L}) \qquad \mathcal{D}_C(v) = \{l\} \qquad\qquad \mathcal{D}_C^{-1} : \mathbb{L} \mapsto \mathbf{Bag}(C) \qquad \mathcal{D}_C^{-1}(l) = v.$$

For each label l introduced by $\mathcal{D}_C$, $\mathrm{s}^\Gamma_{\mathbf{Bag}(C)}$ constructs a dictionary, mapping l to the flat component of the shredded version of v. This is done by first using the dictionary $\mathcal{D}_C^{-1}$, to obtain v and applying s_C^F to shred its contents.

Converting a shredded bag $R^F : \mathbf{Bag}(A^F), R^\Gamma : A^\Gamma$, back to nested form can be done via $\mathbf{for}\ x\ \mathbf{in}\ R^F\ \mathbf{union}\ \mathrm{u}_A[R^\Gamma](x)$, which replaces the labels in R^F by their definitions from the context R^Γ, as computed by $\mathrm{u}_A[a^\Gamma]$ (Figure 5b). We note that the definitions themselves also have to be recursively turned to nested form, which is done in $\mathrm{u}_{\mathbf{Bag}(C)}$.

B.2 Example: Label dictionaries

We give a couple of examples where we contrast the outcome of label unioning dictionaries with that of applying bag addition on them (we use x^n to denote n copies of x).

$$[l_1 \mapsto \{b_1\}, l_2 \mapsto \{b_2, b_3\}] \cup [l_2 \mapsto \{b_2, b_3\}, l_3 \mapsto \{b_4\}] = [l_1 \mapsto \{b_1\}, l_2 \mapsto \{b_2, b_3\}, l_3 \mapsto \{b_4\}]$$
$$[l_1 \mapsto \{b_1\}, l_2 \mapsto \{b_2, b_3\}] \uplus [l_2 \mapsto \{b_2, b_3\}, l_3 \mapsto \{b_4\}] = [l_1 \mapsto \{b_1\}, l_2 \mapsto \{b_2^2, b_3^2\}, l_3 \mapsto \{b_4\}]$$

$$[l_1 \mapsto \{b_1\}, l_2 \mapsto \{b_2, b_3\}] \cup [l_2 \mapsto \{b_5\}, l_3 \mapsto \{b_4\}] = \text{error}$$
$$[l_1 \mapsto \{b_1\}, l_2 \mapsto \{b_2, b_3\}] \uplus [l_2 \mapsto \{b_5\}, l_3 \mapsto \{b_4\}] = [l_1 \mapsto \{b_1\}, l_2 \mapsto \{b_2, b_3, b_5\}, l_3 \mapsto \{b_4\}]$$

As we can see from these examples, bag addition allows us to modify the label definitions stored inside the dictionaries, which is otherwise not possible via label unioning.

B.3 Complexity of Shredding

In this section we show that shredding nested bags can be done in TC_0. By NC_0 we refer to the class of languages recognizable by LOGSPACE-uniform families of circuits of polynomial size and constant depth using and- and or-gates of bounded fan-in. The related complexity class AC_0 differs from NC_0 by allowing gates to have unbounded fan-in, while TC_0 extends AC_0 by further permitting so-called majority-gates, that compute "true" iff more than half of their inputs are true. For details on circuit complexity and the notion of uniformity we refer to [21].

We recall that the standard way of representing flat relations when processing them via circuits is the unary representation, i.e. as a collection of bits, one for each possible tuple that can be constructed from the active domain and the schema, in some canonical ordering, where a bit being turned on or off signals whether the corresponding tuple is in the relation or not. In such a representation (denote by F^{Set} below), if the active domain has size σ, then the number of bits required for encoding a relation whose schema has n_f fields is σ^{n_f}. For example, for a relation with a single field, we need σ bits to encode which values from the active domain are present or not. We also assume a total order among the elements of the active domain, and that the bits of F^{Set} are in lexicographical order of the tuples they represent.

In the case of bags, whose elements have an associated multiplicity, we work with circuits that compute the multiplicity of tuples modulo 2^k, for some fixed k. Thus, for every possible tuple in a bag we use k bits instead of a single one, in order to encode the multiplicity of that tuple as a binary number. In the following we use F^{Bag} to refer to this representation of bags.

For nested values the F^{Set} representation discussed above is no longer feasible since it suffers from an exponential blowup with every nesting level. This becomes apparent when we consider that representing in unary an inner bag with n_t possible tuples requires 2^{n_t} bits. Consequently, for a nested value we use an alternate representation N^{Str}, as a relation $S(p, s)$ which encodes the string representation (over a non-fixed alphabet that includes the active domain, the possible atomic field values) of the value by mapping each position p in the string to its corresponding symbol s.

Example 8. *The string representation $\{\langle a, \{b, c\}\rangle, \langle d, \{e, f\}\rangle\}$, of a nested value of type* $\mathbf{Bag}(Base \times \mathbf{Bag}(Base))$*, is encoded by relation $S(p, s)$ as follows (we show tuples as columns to save space):*

p	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
s	{	⟨	a	,	{	b	,	c	}	⟩	,	⟨	d	,	{	e	,	f	}	⟩	}

For a particular input size n, the active domain of S consists of σ_{ext} symbols including the active domain of the database, delimiting symbols "⟨", "⟩", ",", "{", "}", as well as an additional symbol for each possible position in the string (i.e. $\sigma_{ext} = \sigma + 5 + n$). We remark that the F^{Set} representation of S requires σ_{ext}^2 bits and thus remains polynomial in the size of the input.

This representation may seem to require justification, since strings over an unbounded alphabet may seem undesirable. We note that the representation is fair in the sense that it does not require a costly (exponential) blow-up from the practical string representation that could be used to store the data on a real storage device such as a disk; we use a relational representation of the string and the canonical representation of relations as bit-sequences that is standard in circuit complexity. The one way we could have been even more faithful would have been to start with exactly the bit-string representation by which an (XML, JSON, or other) nested dataset would be stored on a disk. This – breaking up the active domain values into bit sequences – is however avoided for the same reason it is avoided in the case of the study of the circuit complexity of queries on flat relations – reconstructing the active domain from the bit string dominates the cost of query evaluation.

We can now give our main results of this section.

Theorem 10. *Shredding a nested bag from N^{Str} representation to a flat bag (F^{Bag}) representation is in* TC_0.

Proof. To obtain our result we take advantage of the fact that first-order logic with majority-quantifiers (FOM) is in TC_0 [4], and express the shredding of a nested value as a set of FOM queries over the $S(p, s)$ relation.

We start by defining a family of queries $\mathrm{Val}_A(i, j)$ for testing whether a closed interval (i, j) from the input contains a value of a particular type A:

$$\mathrm{Val}_{Base}(i, j) \coloneqq S_{Base}(i) \wedge i = j$$
$$\mathrm{Val}_{A_1 \times A_2}(i, j) \coloneqq S_{\langle}(i) \wedge S_{\rangle}(j) \wedge \exists k.\, \mathrm{Pair}_{A_1, A_2}(i + 1, k, j - 1)$$

$$\begin{aligned}
\text{Pair}_{A_1,A_2}(i,k,j) &:= S_{,}(k) \wedge \text{Val}_{A_1}(i,k-1) \wedge \text{Val}_{A_2}(k+1,j)\\
\text{Val}_{\mathbf{Bag}(C)}(i,j) &:= S_{\{}(i) \wedge S_{\}}(j) \wedge (j = i+1 \vee \text{Seq}_C(i+1,j-1))\\
\text{Seq}_C(i,j) &:= \exists k,l.\, \text{Elem}_C(i,k,l,j) \wedge \forall k,l.\, \text{Elem}_C(i,k,l,j) \rightarrow (\text{EndsWith}_C(i,k) \wedge \text{StartsWith}_C(l,j))\\
\text{Elem}_C(i,k,l,j) &:= (i \le k \wedge l \le j \wedge \text{Val}_C(k,l))\\
\text{EndsWith}_C(i,k) &:= i = k \vee (S_{,}(k-1) \wedge \exists k'.i \le k' \wedge \text{Val}_C(k',k-2))\\
\text{StartsWith}_C(l,j) &:= l = j \vee (S_{,}(l+1) \wedge \exists l'.l' \le j \wedge \text{Val}_C(l+2,l'))
\end{aligned}$$

where $S_{Base}(i)$ is true iff we have a $Base$ symbol at position i in the input string (and analogously for $S_{\{}(i), S_{\}}(i), S_{\langle}(i), S_{\rangle}(i)$ and $S_{,}(i)$). When determining if an interval (i,j) contains a bag value of type $\mathbf{Bag}(C)$ we test if it is either empty, i.e. $j = i+1$ or if it encloses a sequence of C elements (using Seq_C), i.e. it has at least one C element and each element is preceded by another C element or is the first in the sequence, and succeeded by another C element or is the last in the sequence. We use auxiliary queries: $\text{Elem}_C(i,k,l,j)$, which returns true iff the interval (i,j) contains a value of type C between indices k and l, and $\text{StartsWith}_C(l,j)$ / $\text{EndsWith}_C(i,k)$ which returns true iff the intervals (l,j) / (i,k) are either empty or they begin, respectively end, with a value of type C.

For shredding the value contained in an interval (i,j) of the input we define the following family of queries $\text{Sh}^F_A(i,j,p,s)$:

$$\begin{aligned}
\text{Sh}^F_{Base}(i,j,p,s) &:= i \le p \wedge p \le j \wedge S(p,s)\\
\text{Sh}^F_{A_1 \times A_2}(i,j,p,s) &:= \exists k.\, \text{Pair}_{A_1,A_2}(i+1,k,j-1) \wedge (\text{Sh}^F_{A_1}(i+1,k-1,p,s) \ \vee \ \text{Sh}^F_{A_2}(k+1,j-1,p,s))\\
\text{Sh}^F_{\mathbf{Bag}(C)}(i,j,p,s) &:= p = i \wedge s = i,
\end{aligned}$$

where the shredding of bag values results in their replacement with a unique identifier, i.e. the index of their first symbol, that acts as a label. Additionally, the definitions of these labels, i.e. the shredded versions of the bags they replace are computed via:

$$\text{Dict}_C(p,s) := \exists i,j,k,l.\, \text{Val}_{\mathbf{Bag}(C)}(i,j) \wedge \text{Elem}_C(i+1,k,l,j-1) \wedge ((p = k-1 \wedge s = i) \vee \text{Sh}^F_C(k,l,p,s)),$$

where we prepend to each shredded element in the output the label of the bag to which it belongs (we can do that by reusing the index of the preceding "{" or "," present in the original input). We build a corresponding relation Dict_C for every bag type $\mathbf{Bag}(C)$ occurring in the input. These relations encode a flat representation of the input, as bags of type $\mathbf{Bag}(\mathbb{L} \times C^F)$, where each tuple uses a fixed number of of symbols, therefore we no longer make use of delimiting symbols.

For our example input, we only have two bag types, $\mathbf{Bag}(Base \times \mathbf{Bag}(Base))$ and $\mathbf{Bag}(Base)$, and their corresponding relations are:

$\text{Dict}_{Base \times \mathbf{Bag}(Base)}(p,s) :=$

p	1	3	5	11	13	15
s	1	a	5	1	d	15

$\text{Dict}_{Base}(p,s) :=$

p	5	6	7	8	15	16	17	18
s	5	b	5	c	15	e	15	f

.

The flat values that they encode are $\{\langle 1,a,5\rangle, \langle 1,d,15\rangle\} : \mathbf{Bag}(\mathbb{L} \times Base \times \mathbb{L})$ and $\{\langle 5,b\rangle, \langle 5,c\rangle, \langle 15,e\rangle, \langle 15,f\rangle\} : \mathbf{Bag}(\mathbb{L} \times Base)$.

However, the Dict_C relations cannot be immediately used to produce the sequence of tuples that they encode since the indices p associated with their symbols are non-consecutive. To address this issue we define:

$$\text{ToSeq}[X](p',s) := \exists p.X(p,s) \wedge p' = \#u(\exists w.X(u,w) \wedge u \le p),$$

which maps each index p in relation $X(p,s)$ to an index p' corresponding to its position relative to the other indices in X. To do so we used predicate $p' = \#u\Phi(u)$ to count the number of positions u for which $\Phi(u)$ holds, since it is expressible in FOM [4].

Finally, we determine the shredded version of an input value $x : \mathbf{Bag}(B)$, based on its N^{Str} representation $S(p,s)$, as $S^F(p,s) := \text{ToSeq}[\text{Dict}_B(p,s) \wedge s \neq 1]$ where we filter out from $\text{Dict}_B(p,s)$ those symbols denoting that a tuple belongs to the top level bag, identified by label 1. The shredding context is defined by a collection of relations $S^\Gamma := \text{Sh}^\Gamma_B$, where:

$$\text{Sh}^\Gamma_{Base} := \varnothing \qquad \text{Sh}^\Gamma_{A_1 \times A_2} := \langle \text{Sh}^\Gamma_{A_1}, \text{Sh}^\Gamma_{A_2} \rangle \qquad \text{Sh}^\Gamma_{\mathbf{Bag}(C)} := \langle \text{ToSeq}[\text{Dict}_C], \text{Sh}^\Gamma_C \rangle$$

The last step that remains is to convert the resulting flat bags from the current representation (as $X(p,s)$ relations in F^{Set} form) to the F^{Bag} representation. We recall that each such relation encodes a sequence of tuples such that each consecutive group of n_f symbols (according to their positions p) stands for a particular tuple in the bag, where n_f is the number of fields in the tuple. Additionally, since the bits in the F^{Set} representation are lexicographically ordered it follows that each consecutive group of σ_{ext} bits contains the unary representation of the symbol located at that position. Therefore, we can find out how many copies of a particular tuple t are in the bag by counting (modulo 2^k) for how many groups of $n_f \cdot \sigma_{ext}$ bits we have unary representations of symbols that match the symbols in t. By performing this counting for all possible tuples t in the output bag we obtain the F^{Bag} representation of $X(p,s)$. We note that both testing whether particular bits are set and counting modulo k are in TC_0.

Since $S^F(p,s)$ and S^Γ can be defined via FOM queries, and since their conversion from $X(p,s)$ relations in F^{Set} form to the F^{Bag} representation uses a TC_0 circuit, this concludes our proof that shredding nested values from N^{Str} to F^{Bag} representation can be done in TC_0. $\square$