Compilation Techniques for Incremental Collection Processing

THÈSE N^{O} 8019 (2017)

PRÉSENTÉE LE 27 OCTOBRE 2017
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE THÉORIE ET APPLICATIONS D'ANALYSE DE DONNÉES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Daniel LUPEI

acceptée sur proposition du jury:

Prof. W. Zwaenepoel, président du jury
Prof. C. Koch, directeur de thèse
Prof. V. Tannen, rapporteur
Prof. M. Zaharia, rapporteur
Prof. V. Kuncak , rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2017

All things belong to the same order of things, for such is the oneness
of human perception, the oneness of individuality, the oneness of
matter, whatever matter may be.
The only real number is one, the rest are mere repetition.
— Vladimir Nabokov

To my extended family comprising of mum, dad, my sister,
Marina, Delia and Adrian

Acknowledgements

First, I would like to thank my advisor Christoph Koch for all the support he has given me throughout my PhD journey. He has encouraged me in pursuing my curiosity and developing extensive expertise in topics that were very satisfying for me intellectually and allowed me to develop both a broad as well as deep understanding of my research field. Without his strong and sustained guidance and encouragements none of my phd work would have been possible. Moreover, he has done all in his power for me to get as much exposure as possible to both academic and industrial research, by facilitating visits with professors from other universities or having me do internships in research labs. I also consider myself privileged to have benefited from his unique point of view towards challenges in the field of databases as well as his advice wrt. approaching research problems.

I would like to thank the members of my thesis committee who kindly accepted to offer their time and energy to assess my dissertation and to suggest improvements. In particular, I would like to thank Val Tannen who has generously guided my first steps in crafting and presenting a research contribution. His advice has stuck with me for the duration of my PhD, and he has continuously been someone I could turn to for advice and support. Furthermore, much of my work would not have been possible but for the work he has done before in advancing the state-of-the-art in query languages for the nested data model. I would like to thank Todd Mytkowicz and Madan Musuvathi for mentoring me during my internship at Microsoft Research, and to all the other members of the group with whom I was fortunate enough to collaborate.

I would like to thank to all the members of the DATA lab for making my PhD journey an enjoyable one. I thank Milos Nikolic for his help each time I was seeking an opinion on my work, and his friendship as we were both doing an internship in Microsoft. I thank Andrej Spielmann, Thierry Coppey, Immanuel Trummer and Sachin Basil John, my office mates, for tolerating me over the years and for their patience in listening to my research conundrums. I thank Yannis Klonatos, Aleksandar Vitorovic, Mohammed ElSeidy, Mohammad Dashti, Amir Shaikhha and Lionel Parreaux for the invaluable discussions, insightful comments, and the constructive feedback on my ongoing research and on my presentations. I would like to thank Simone Muller for helping me in all the administrative problems I faced, and for making sure that everything ran smoothly within our lab.

Thank you to Adrian Popescu, Marina Boia and Delia Stancu for the unforgettable time spent together in the past six years, and for making sure I would never feel alone or uncared for. My experience as a PhD student would be just a mere sketch but for the color they brought to every outing, coffee break or trip we made. Each of you have changed me in countless ways, taught me how to persevere, to find laughter and beauty, and to trust in my abilities. Your example continues to inspire me every day. I have greatly enjoyed the friendship of Valentina Sintsova, Irina and Eugen Paraschiv, Corina and Catalin Trif, Cristina Radu, Alina Dudeanu, Ina Bialova, Yannis Klonatos, Mohammed ElSeidy, Irina Prostakova, Andrew Becker, Sergii Vozniuk, Eleni Tzirita Zacharatou, Mirjana Pavlovic, Cristina Ghiurcuta, Danica Porobic, Mihai Martalogu, Florin Dinu, Gregoire Devauchelle, Onur Yürüten. Having someone to share my worries with and having them support and comfort me through the tough times has made all the difference. I will always remember fondly our time cooking pancakes or delicious cakes, hiking or skiing over the Alps, or the late hours picnics by the lake.

I would like to thank my family for their unconditional moral support and love.

Lausanne, September 2017 D. L.

Abstract

Many map-reduce frameworks as well as NoSQL systems rely on *collection* programming as their interface of choice due to its rich semantics along with an easily parallelizable set of primitives. Unfortunately, the potential of collection programming is not entirely fulfilled by current systems as they lack efficient incremental view maintenance (IVM) techniques for queries producing large nested results. This comes as a consequence of the fact that the nesting of collections does not enjoy the same algebraic properties underscoring the optimization potential of typical collection processing constructs.

We propose the first solution for the efficient incrementalization of collection programming in terms of its core constructs as captured by the positive nested relational calculus (NRC^+) on bags (with integer multiplicities). We take an approach based on *delta* query derivation, whose goal is to generate delta queries which, given a small change in the input, can update the materialized view more efficiently than via recomputation. 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, can also be applied to IncNRC^+ .

Existing systems are also limited wrt. the size of inner collections that they can effectively handle before running into severe performance bottlenecks. In particular, in the face of nested collections with skewed cardinalities developers typically have to undergo a painful process of manual query re-writes in order to ensure that the largest inner collections in their workloads are not impacted by these limitations.

To address these issues we developed SLeNDer, a compilation framework that given a nested query generates a set of semantically equivalent (partially) *shredded* queries that can be efficiently evaluated and incrementalized using state of the art techniques for handling skew and applying *delta* changes, respectively. The derived queries expose nested collections to the same opportunities for distributing their processing and incrementally updating their contents as those enjoyed by top-level collections, leading on our benchmark to up to 16.8x and 21.9x speedups in terms of offline and online processing, respectively.

Abstract

In order to enable efficient IVM for the increasingly common case of collection programming with functional values as in Links, we also discuss the efficient incrementalization of simply-typed lambda calculi, under the constraint that their primitives are themselves efficiently incrementalizable.

Key words: collection programming, incremental computation, incremental view maintenance, delta processing, higher-order incrementalization, nested relational calculus, shredding, simply-typed lambda calculus

Abstract

De nombreux map-reduce frameworks ainsi que des systèmes NoSQL s'appuient sur la programmation de *collecte* comme leur interface de choix en raison de sa sémantique enrichie ainsi que d'un ensemble de primitives facilement parallélisable. Malheureusement, le potentiel de la programmation de collecte n'est pas entièrement rempli par les systèmes actuels car ils manquent de techniques efficaces de maintenance incrémentielle de vue (MIV) pour les requêtes produisant de gros résultats imbriqués. Cela vient en conséquence du fait que l'imbrication des collections ne bénéficie pas des mêmes propriétés algébriques soulignant le potentiel d'optimisation des primitives de traitement typiques de collecte.
Nous proposons la première solution pour le incrementalization efficace du calcul relationnel imbriqué positif (CRI$^+$) sur les sacs (avec multiplicités entier), car ses opérateurs représentent le cœur de la programmation de la collecte. Nous prenons une approche basée sur la dérivation de requête *delta*, dont l'objectif est de générer des requêtes delta qui, compte tenu d'un petit changement dans l'entrée, peuvent mettre à jour la vue matérialisée plus efficacement que par recomputation. Plus précisément, nous modélisons le coût des opérateurs du CRI$^+$ et classifions les requêtes comme efficacement incrémentalisable, si leur delta a un coût strictement inférieur à celui de la réévaluation complète. Ensuite, nous identifions IncCRI$^+$, un grand fragment de CRI$^+$ qui est efficacement incrémentalisable et nous fournissons une transformation qui préserve la sémantique alors qu'elle convertit toute requête CRI$^+$ en une collection de requêtes IncCRI$^+$. En outre, nous prouvons que la maintenance incrémentielle de CRI$^+$ se situe dans la classe de complexité NC$_0$ et nous montrons comment l'MIV récursif, une technique qui a fourni des accélérations significatives par rapport à l'MIV traditionnel dans le cas de requêtes plates, peut également être appliquée à IncCRI$^+$.

Les systèmes existants sont également limités par rapport à la taille des collections internes qu'ils peuvent gérer efficacement avant de rencontrer des goulots d'étranglement sérieux. En particulier, face à des collections imbriquées avec de cardinalités asymétriques les développeurs ont généralement subir un processus douloureux de réécritures de requête manuelle afin d'assurer que les plus grandes collections intérieures de leur charge de travail ne sont pas affectées par ces restrictions.
Pour résoudre ces problèmes, nous avons développé SLeNDer, un framework de compilation qui a partir d'une requête imbriquée génère un ensemble de requêtes sémantiquement équivalentes (partiellement) déchiquetées qui peuvent être évaluées et incrémentées de manière efficace en utilisant des techniques d'art pour le traitement des biais et l'application des modifications *delta*, respectivement. Les requêtes dérivées exposent les collections imbriquées aux

mêmes opportunités pour distribuer leur traitement et mettent à jour de façon incrémentielle leurs contenus que ceux dont bénéficient les collections de premier niveau, ce qui conduit pour notre benchmark à des accélérations allant jusqu'à 16.8x et 21.9x en termes de traitement hors ligne et en ligne respectivement.

Afin de permettre une MIV efficace pour le cas de plus en plus fréquent de la programmation de collecte avec des valeurs fonctionnelles comme dans Links, nous discutons également de l'incrémentalisation efficace des lambda-calculs simplement typés, sous la contrainte que leurs primitives sont efficacement incrémentables.

Mots clefs: Programmation de collecte, calcul incrémentiel, maintenance incrémentielle, traitement delta, l'incrémentation d'ordre supérieur, le calcul relationnel imbriqué, le déchiquetage, lambda-calcul simplement typé

Contents

1 Introduction

Large scale data processing has become indispensable in many areas of computer science. For example, in infrastructure management having the capability to analyze massive amounts of data can provide valuable insights into complex systems and help reduce their operating costs. Similarly, since the results of learning algorithms, such as those powering recommender systems, can only be as good as the data they are being fed, the larger and richer the input dataset, the better their outcome. In fact, the significant improvements in the accuracy of such algorithms due to the analysis of ”big data“ have fueled a renaissance in machine learning and artificial intelligence.

The need to process ever larger amounts of data has come at odds with the ending of Moore’s Law, which for the past decades saw the number of transistors per chip double every year, with a corresponding gain in processing power. Currently, one can no longer simply rely on processor upgrades in order to handle the expanding workloads. This has made parallelization a core concern in the design of algorithms and has prompted the development of map-reduce frameworks (e.g. Apache Spark [72], Apache Pig [55], Scope/Cosmos [14]) which provide programmers with familiar abstractions over cluster resources, while relieving them from the many headaches of distributed computing, like resource allocation, scheduling, or faults in the underlying infrastructure.

Since a parallel application can only be as fast as its slowest component, map-reduce frameworks must make sure to distribute workloads as evenly as possible across processing nodes. To do so they rely on *collection programming* as it provides a compelling tradeoff between ease of parallelization and expressive power, i.e. its primitives are embarrassingly parallel (eg. map, filter), while being expressive enough to support a variety of analytical tasks, like machine learning or graph processing. Moreover, its close connection to classic relational querying languages makes it possible to benefit from decades of database research on query optimizations.

1.1 Why collection programming?

Collection programming has been widely adopted as the interface of choice by many map-reduce frameworks and NoSQL systems due to its desirable combination of object-oriented and functional (higher-order) features of modern programming languages within a rich algebraic framework common to standard querying languages like SQL. Its roots can be traced back to calculi for complex/nested values and object-oriented querying languages, whose relaxed data models allow for collections to be nested within records. Throughout the thesis we rely on particular variants of nested relational calculi as formalizations for the core of collection programming.

The shared algebraic framework with classic relational calculi results in collection programming being similarly embarrassingly parallelizable. Indeed, certain versions of nested relational calculi have been shown to reside in a similarly low parallel complexity class [66] (NC vs. AC_0 for the flat relational calculus). This is especially relevant in the context of map-reduce frameworks, as it indicates that queries written in such languages can indeed be uniformly scaled out. Unfortunately, current systems do not fully realize this potential for parallelization as they only distribute processing at the granularity of top-level records, while the computation of inner collections is performed sequentially. They are thus vulnerable to load imbalance, whenever the sizes of inner collections differ significantly from one top-level record to the next, or when top-level collections have low-cardinality (which can happen is they are the result of grouping wrt. columns whose domain is small).

The embedding of collection programming withing general-purpose functional programming languages offers a host of advantages from an expressiveness and a software development perspective. Being able to work with a wide range of datatypes combined with the ability to easily integrate third-party libraries, considerably simplifies the development of complex analytical pipelines across a variety of domains. Just having records that can hold nested collections already opens up the possibility of representing hierarchical relationships in a natural way, and eliminates the need for normalization when ingesting data or joining foreign keys when processing it. Besides simpler and more maintainable code, the removal of these steps offers a considerable boost in performance as they both require expensive data reshuffling operations.

These advantages have also underlined the proposal of language integrated querying frameworks based on collection programming (eg. LINQ, Ferry, Links) that aim at providing a single programming environment for all the layers of a system (from the database level to the frontend) [1].

[1] It is in fact not uncommon in industry to have the data management components of a system being developed in JavaScript, which underscores the need for a unified programming environment for multi-tiered system development.

1.2 Incremental evaluation

Considering the large scale of data being processed it is rarely feasible to re-execute an entire workload every time new data becomes available. Therefore, large-scale collection processing in frameworks such as Spark [72] can greatly benefit from incremental maintenance in order to minimize query latency in the face of updates. In addition, the monetization of processing resources within cloud platforms makes it harder to ignore the waste associated with re-evaluation, while manually creating and maintaining views in order to avoid recomputing them is hard to scale to workloads consisting of complex, rapidly evolving query sets.

Streaming engines partially address this issue as they continuously update the output of queries on incoming data. However, since they only provide limited semantics to their operators (eg. window semantics), developing applications on top of them becomes challenging as soon as the underlying logic requires joining data from multiple sources.

By contrast, incrementalization techniques that speed up the propagation of input changes to the output based on materializing intermediate results do so while preserving semantics. Among such techniques, the use of *delta queries* to perform incremental view maintenance (IVM) 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. Its success is based on the fact that applying updates by delta queries requires little support at the query engine level. In addition, the derived delta queries can be further optimized within the same optimization framework as the original queries.

Delta derivation leverages the algebraic properties of operators to split the definition of a view into two parts, one that only depends on the original input and thus can be materialized and reused, and another, i.e. the delta query, that also depends on the update, but is potentially cheaper to evaluate. For example, consider a query that computes a sum aggregate over a column of an input relation. Then, if some new tuples are added to the input in the form of a delta relation, the updated view definition can be expressed as the addition between a query that aggregates over the original input and a second query that aggregates over the delta relation. Since we already have the result of the first query, updating the view then requires evaluating only the latter, which is considerably cheaper in the case of incremental updates. In many cases deltas are actually asymptotically faster – for instance, the original query in the previous example 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). We expand on delta-based incrementalization techniques with a more detailed example in Section 2.2.

The benefits of incremental maintenance can be amplified if one applies it recursively [35] – one can also speed up the evaluation of delta queries 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}) [35], and outperforms classical IVM by many orders of magnitude [36]. However, the techniques described above target only flat relational queries and as such cannot be used to enable incremental maintenance for collection processing engines.

Unfortunately, delta processing is strongly dependent on the semantics of the target query language as well as on the type of changes it can apply. In particular, the incrementalized primitives must enjoy a rich set of algebraic re-writings wrt. to the update operations. While these conditions are largely met by classic querying languages over flat relations, satisfying them in the context of more expressive languages has proven problematic.

1.3 Challenges to incrementalizing nested queries

In the context of the incremental maintenance of views (IVM), a nested data model has negative performance implications as one has to choose between *deep* or *shallow* incrementalization techniques, i.e. those that support small changes to inner collections, from those that can only handle insertions/deletions of top-level records. However, both face significant drawbacks as we detail in the following, which have prevented them from gaining the same wide adoption enjoyed by online processing techniques targeting flat relational workloads.

Deep incrementalization proposals rely primarily on runtime change propagation algorithms [22, 49, 52], which apply input updates through materialized intermediate results all the way up to the maintained view. These techniques have to perform expensive bookkeeping in order to track the lineage of output values back to the input relations, and thus be able to adjust them according to the input changes. Moreover, by doing so at runtime the optimizations they can employ are restricted only to the physical level.

By contrast, compile-time approaches like the derivation of *delta* queries open up many more opportunities for optimizations (e.g. query factorization, code specialization, etc.). Given a query and an input update primitive, delta processing derives a delta query that computes the corresponding update of the result. While it has proven to be an extremely effective technique for the incremental maintenance of flat queries, current proposals targeting nested workloads provide only *shallow* incrementalization [25].

To understand why, we recall that the delta derivation process relies on per query operator re-write rules that translate the effects of an input update primitive into an output update operation. This translation is predicated on the semantics of the operators involved, which presents delta processing with a tradeoff between the expressiveness of the update primitives supported vs. the class of query operators whose deltas are cheaper than re-evaluation. On one hand, one could use only full tuple insertions/deletions as update primitives and be able to derive efficient deltas for a large class of queries (as in [31]), considering that the interaction of these updates wrt. the other querying constructs is extremely well behaved. On the other hand, an expressive update language could be allowed, but then have a greatly reduced class of views that can be efficiently maintained. For example, Ceri et al. [12] support all of SQL's update primitives but can incrementally update only views that do not use union and whose attributes functionally determine keys of the base relations.

Unfortunately, in the context of nested queries both of the alternatives above, that is the use of full tuple insertions/deletions as update primitives or the use of a more expressive update language, are highly undesirable. The former one forces us to model even small changes to inner collections by a deletion followed by the insertion of an entire top tuple (with its full inner collection updated). Considering that nested collections can be arbitrarily large, such an approach would greatly diminish the benefits of incrementalization. While more expressive update languages that are capable of defining small updates to inner collections have been proposed in the literature [43], they do so in a manner similar to SQL's update primitives (i.e. via selection predicates) which makes them vulnerable to similar shortcomings (as in [12]) wrt. delta derivation.

1.4 Summary of contributions

In this thesis we advance the state of the art wrt. to the incrementalization and scaling of collection programs as follows:

- We propose a delta-based solution for the *deep* incrementalization of collection programs. In order to assess its efficiency we define a cost semantics for collection operators and establish that indeed the incrementalized version of a query is cheaper than re-evaluating it on every update. Furthermore, we prove that the delta queries we derive are in a lower parallel complexity class than the original queries (NC_0 vs TC_0). (Chapter 3)

  In addition, we show that the embedding of collection programming within functional languages is efficiently incrementalizable as well. We do so by proving that, given a set of efficiently incrementalizable primitives, the simply-typed lambda calculus built on top of them is also efficiently incrementalizable. This represents an essential step towards a *static* (i.e. re-writing based) solution for the incrementalization of general-purpose programming languages. (Chapter 4)

- We propose a compilation framework that given a collection program can produce an optimized (Spark) trigger program for incrementalizing its results, as well as generate a semantically equivalent variant that exposes the full parallelization potential of nested queries. It does so by decoupling the processing of inner collections from the top-level collection, such that their computation can be evenly distributed across cluster resources, as opposed to being performed sequentially by the node evaluating its corresponding top-level record. This allows us to avoid the unfortunate scenario where a few skewed inner bags significantly delay the completion of an entire map-reduce job. (Chapter 5)

We incrementalize collection programs over nested data starting from a variant of nested relational calculus (NRC) that clearly separates the flat features of the language from those that can introduce nesting. This allows us to focus our efforts on efficiently incrementalizing the latter ones, while for the former we rely on existing approaches for their delta processing. As this calculus captures the core of collection programming, the solution we propose is immediately applicable to all systems that use collection programming as their main querying interface.

To prove the effectiveness of our approach we formally 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. In addition, we show how the delta processing of nested queries can be further optimized using Recursive IVM [35], a technique that in the case of flat queries has achieved significant speedups over classic IVM [36].

Proving that the incrementalization of collection programs is in a lower parallel complexity class than their evaluation (NC_0 vs. TC_0) is especially relevant in the context of map-reduce frameworks, as it indicates that incrementally updating the results of nested queries can be done with less communication overhead as was required when computing the initial result, considering that an important distinction between NC_0 and TC_0 lies in NC_0's restriction to only use circuits whose gates have bounded fan-in.

We implemented our approaches within SLeNDer [2], a compilation framework providing an abstraction layer for scalable and incrementable nested collection processing. It achieves that by turning nested queries into a semantically equivalent series of *shredded* queries corresponding to the (inner) collections we are interested in parallelizing / incrementalizing. It does so by replacing inner collections with *labels*, and separately maintaining dictionaries mapping labels to their defining bags. By making inner collections independently addressable, labels serve an essential role in being able to update them as well as distribute their processing across nodes.

Our experimental evaluation shows that *recursive* incrementalization on top of shredding results in significant speedups in the refresh rate of views when compared to standard incrementalization techniques as well as full re-evaluation. Moreover, the derived shredded queries

[2] Skew-Less Nested Data

achieve better load balancing when operating on nested collections with skewed cardinalities or on top level collections with low cardinality. We also highlight the effectiveness of partial shredding in reducing the overheads of shredding while retaining most of its benefits.

This work includes material from several publications for which the author of this thesis is the lead author or a co-author.

- Christoph Koch, Daniel Lupei, Val Tannen
  Incremental View Maintenance for Collection Programming
  PODS 2016.
- Daniel Lupei, Milos Nikolic, Christoph Koch
  SLeNDer: Query Compilation for Agile Collection Processing
  Under submission.

2 Background

In this chapter we introduce the version of Nested Relational Calculus that we use for modeling the core constructs of collection programming. Then we recall how delta processing functions in the case of flat relational queries and discuss the challenges for efficient delta-based incrementalization techniques (Sections 2.2 and 2.3). Finally, in Section 2.4 we introduce the standard parallel complexity classes and detail how the operators of relational algebra (which is equivalent to a large fragment of our calculus) fit within them.

2.1 Nested Relational Calculus

In the following we describe our main formalism for collection programming as a variant of Nested Relational Calculus (NRC$^+$) [10, 39, 69, 70] with bag semantics. 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 [35, 42]. In addition, for every bag type we have an empty bag constructor $\emptyset$, 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 $\emptyset$, 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

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

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

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

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

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

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

Figure 2.1 – Typing rules for the nested relational calculus (NRC$^+$).

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. The value assignments of variables are represented by γ and ε, 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.

The typing rules of NRC$^+$ are given in Figure 2.1, 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. Figure 2.2 presents the semantics of NRC$^+$, where $\gamma(X)$ and $\varepsilon(x)$ return the valuation of variables X and x from their respective contexts, we use $\{\cdots\}$ to denote bag values, and $\biguplus_{v \in [[e]]}$ to denote the iterated union over all the bag values in the result of e.

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 bags can be used to simulate negation [10]. We discuss in Section 2.3 the challenges posed by negation wrt. efficient maintenance within our framework.

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

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

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

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

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

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

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

Figure 2.2 – Semantics of the nested relational calculus (NRC$^+$).

Example 1. *Filtering an input bag R according to some predicate p can be defined in* NRC$^+$ *as:*

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

considering that the **for** *construct with* **where** *clause can be expressed as follows:*

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

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

Compared to the standard formulation given in [10] 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 **sng**(e). Therefore we control carefully how singletons are constructed (note that we have four rules for singletons but they do not "overlap"), and we have a separate flattening construct. In addition, we have a bag (Cartesian) product construct instead of a pairing construct, as this simplifies the shredding transformation we propose in the next chapter. These particularities of our version are just cosmetic as we can still express the same class of queries of bag output type as in [10].

Finally, we remark that the **sng**(e) and **flatten** constructs are the only ones that can alter the nesting structure of a given input value, i.e. add or remove nesting levels. Consequently, by discarding them we end up with a language which is equivalent to flat relational calculus, for which standard techniques of delta processing, like those discussed in the following section, are immediately applicable. Therefore, in incrementalizing NRC$^+$ we mainly need to focus our efforts on the delta processing of these two additional constructs, a job made easier by the fact that **flatten** is linear wrt. bag union. It is thus by careful language design that we were able to reduce our problem of incrementalizing NRC$^+$ to the well studied problem of incrementalizing flat relational calculus as well as isolate the difficulties introduced by nesting to a single construct of the language, **sng**(e).

2.2 Delta processing for the flat relational case

We recall how delta processing works for queries expressed in the *positive relational algebra*. Delta rules were originally defined for datalog programs [30,31] but they are even more natural for algebraic query languages such as the relational algebra on bags [27,35], simply because the algebraic structure of a group is the necessary and sufficient environment in which deltas live.

Consider relational algebra expressions built from table names $R_1,\ldots,R_n$ from some schema and the operators for selection σ_p, projection $\Pi_{\bar{i}}$, Cartesian product $\times$, and union, where we denote the last one by $\uplus$ to remind us that we assume bag semantics in this work.

The delta rules constitute an inductive definition of a transformation that maps every algebra expression e over table names R_i into another algebra expression $\delta(e)$ over table names R_i and ΔR_i, $i=1..n$. The names of the form ΔR_i designate an update: tables that contain tuples to be added to those in R_i (for the moment we focus only on insertions). We use δ for the name of the transformation itself. The rules are:

$$\begin{aligned} \delta(R_i) &= \Delta R_i \quad i=1..n & \delta(\sigma_p e) &= \sigma_p \delta(e) \\ \delta(e_1 \uplus e_2) &= \delta(e_1) \uplus \delta(e_2) & \delta(\Pi_{\bar{i}} e) &= \Pi_{\bar{i}} \delta(e) \\ \delta(e_1 \times e_2) &= \delta(e_1) \times e_2 \uplus e_1 \times \delta(e_2) \uplus \delta(e_1) \times \delta(e_2) \\ \delta(e) &= \varnothing \quad \text{(when no } R_i \text{ occurs in } e\text{).} \end{aligned}$$

We remark that the rule for join is the same as the one for Cartesian product.

The delta rules satisfy the following property, which also suggests how the incremental computation proceeds:

$$e[R_1 \uplus \Delta R_1, \cdots, R_n \uplus \Delta R_n] \;=\; e[R_1, \cdots, R_n] \uplus \delta(e)[R_1, \cdots, R_n, \Delta R_1, \cdots, \Delta R_n] \tag{2.1}$$

This is due to the commutativity and associativity of bag union as well as the distributivity of selection, projection and Cartesian product, over bag union.

In the statement above we abuse, as usual, the notation by using the R_i's for both table names and corresponding table instances and we denote by $e[\overline{R}]$ the table that results from evaluating the algebra expression e on a database $\overline{R}$, where $\overline{R}$ stands for $R_1,\ldots,R_n$. Equation (2.1) captures the incremental maintenance of the query result. Given updates $\overline{\Delta R}$ to the database, we just compute $\delta(e)[\overline{R},\overline{\Delta R}]$ and use it to update the previously materialized answer $e[\overline{R}]$.

Example 2. *For a concrete example of incrementalizing a relational algebra query, we consider a bag of movies $M(movie, genre)$, a bag containing their showtimes $Sh(movie, location, time)$ and the query DOz returning all the dramas playing in Oz:*

$$DOz \;\equiv\; \Pi_{movie}(\sigma_{location=\mathrm{Oz}} Sh \bowtie \sigma_{genre=\mathrm{Drama}} M).$$

Now suppose that the updates ΔSh and ΔM are applied to Sh and M, respectively. By Equation 2.1 and the delta rules, the updated DOz can be computed by $\uplus$-ing

$$\begin{aligned}\Pi_{movie}(&\sigma_{location=\mathrm{Oz}} \Delta Sh \bowtie \sigma_{genre=\mathrm{Drama}} M\\ &\uplus\ \sigma_{location=\mathrm{Oz}} Sh \bowtie \sigma_{genre=\mathrm{Drama}} \Delta M\\ &\uplus\ \sigma_{location=\mathrm{Oz}} \Delta Sh \bowtie \sigma_{genre=\mathrm{Drama}} \Delta M)\end{aligned}$$

to the previously materialized answer to DOz. If ΔSh and ΔM are much smaller than Sh, respectively M, this is typically computationally much cheaper than recomputing the query after updating the base tables: this is what makes incremental view maintenance worthwhile.

Under reasonable assumptions about the cost of query evaluation algorithms and considering small updates compared to the size of the database, this is better than recomputing the query on the updated database $e[\overline{R \uplus \Delta R}]$. For instance, a query $R \bowtie S$ can have size (and evaluation cost) quadratic in the input database. Assuming ΔR and ΔS consist of a constant number of tuples, incrementally maintaining the query via $\delta(R \bowtie S) = (\Delta R) \bowtie S \uplus R \bowtie (\Delta S) \uplus (\Delta R) \bowtie (\Delta S)$ has linear size and cost, while recomputing it (as $(R \uplus \Delta R) \bowtie (S \uplus \Delta S)$) has quadratic cost.

As shown by Gupta et al. [30], the same delta rules can also be used to propagate deletions if we extend the bag semantics to allow for *negative* multiplicities: the table ΔR_i associates negative multiplicities to the tuples to be deleted from R_i.

2.3 Efficient delta processing

In the following we discuss the difficulties 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 re-computation (or simply *efficient*), if for any update ΔR s.t. size$(\Delta R) \ll$ 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.:

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

where the second equation ensures that applying the update is also 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:

$$\begin{aligned}&\text{cost}(\delta(\mathbf{p}(e))[R,\Delta R]) \ll \text{cost}(\mathbf{p}(e)[R \uplus \Delta R]) \;\text{ and}\\&\text{size}(\delta(\mathbf{p}(e))[R,\Delta R]) \ll \text{size}(\mathbf{p}(e)[R \uplus \Delta R])\end{aligned} \tag{2.2}$$

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 $\text{cost}(\delta(\mathbf{p}(e))[R,\Delta R]) > \text{cost}(e[R])$), as it leads to the following contradiction:

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

where the last approximation follows from the fact that:

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

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. selection σ_p).

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 from Chapter 3 for efficiently incrementalizing the singleton constructor $\mathbf{sng}(\cdot)$, we take advantage of the fact that the only way of accessing the contents of a inner bag is via the flattening operator $\mathbf{flatten}(\cdot)$.

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 (2.2) can never be satisfied. This problem extends to a class of primitives that includes bag equality, negation, and membership testing, and restricts our approach for efficient incrementalization to only the positive fragment of collection programming.

2.4 Parallel complexity classes

As opposed to standard complexity classes which characterize the time/space needed by a given algorithm when executing on a Turing machine, parallel or circuit complexity classes look at the requirements of problems when implemented as dedicated circuits and the amount of hardware available is essentially unconstrained. It aims at evaluating the difficulty of a problem under the assumption that the number of processing units can be made proportional to the size of the input, thus highlighting its behavior on an idealized model of a parallel machine. This helps provide a theoretical limit for the improvements that can be achieved by "throwing" more hardware at a problem, capturing its intrinsic scalability. In particular, problems that are embarrassingly parallel are expected to fit within low parallel complexity classes.

Since in cluster computing the emphasis shifts more from designing fast algorithms to coming up with highly scalable approaches, understanding the parallel complexity of a problem is essential as it provides strong bounds on the maximum parallelism that can be achieved by any particular solution of that problem, given enough hardware resources. Moreover by understanding the circuit architecture required for solving the problems in a specific parallel complexity class one can then reverse engineer it in order to map it to more common hardware or runtime environments, and thus come up with solutions that fully exploit their intrinsic parallelization potential.

From the perspective of incrementalization we deem it essential that any proposed approach fits within either the same or a lower parallel complexity class when compared to the target language.

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 (denoted by F^{Set}), if the active domain has size m, then the number of bits required for encoding a relation whose schema has n_f fields is m^{n_f}. For instance, for a relation with two fields, we need m^2 bits to encode which tuples are present or not (a concrete example is presented in Figure 2.3). 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

G(a,a) ∈ R
G(a,b) ∈ R
G(b,a) ∈ R
G(b,b) ∈ R

Figure 2.3 – The gates representing a binary relation $R(x, y)$ when the active domain consists of only two values, a and b.

as a binary number. We argue that having a binary representation where k is fixed for example to 128 satisfies most common practical scenarios. In the following we use F^{Bag} to refer to this representation of bags.

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. That is, NC_0 represents the class of problems that given enough hardware can be solved in constant time. More generally, NC_k extends the depth constraint to $O\big(\log^k(\cdot)\big)$ while $\mathrm{NC} := \bigcup_k \mathrm{NC}_\mathrm{k}$, a.k.a. Nick's class [1], is considered the class of "highly" parallelizable problems. Furthermore, Stockmeyer and Vishkin [62] have related the NC class to the class of functions computable by a *Concurrent Read Concurrent Write Parallel Random Access Machine* in polylogarithmic time using polynomially many processors.

The related complexity class AC_k differs from NC_k by allowing gates to have *unbounded* fan-in, while TC_k extends AC_k by further permitting so-called majority-gates, that compute "true" iff more than half of their inputs are true. The distinction between NC_k on one hand, and $\mathrm{AC}_\mathrm{k}/\mathrm{TC}_\mathrm{k}$ on the other, in terms of allowing gates with unbounded fan-in is especially relevant when it comes to real hardware, since unbounded fan-in typically implies an expensive communication step.

Moreover, these parallel complexity classes can be placed in the following hierarchy:

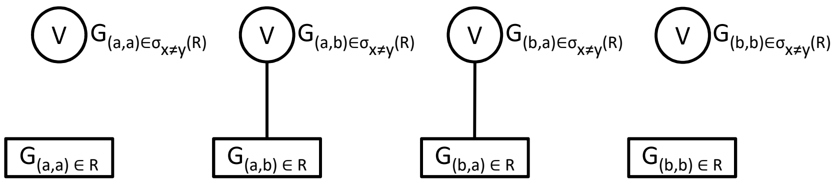
$$\begin{aligned}\mathrm{AC}_0 \subseteq \mathrm{TC}_0 \subseteq \mathrm{NC}_1 \subseteq \mathrm{LOGSPACE} \subseteq \mathrm{AC}_1 \subseteq \cdots \subseteq \mathrm{AC}_\mathrm{i} \subseteq \mathrm{TC}_\mathrm{i} \subseteq \mathrm{NC}_{\mathrm{i}+1} \subseteq \cdots \\ \subseteq \mathrm{NC} = \bigcup_k \mathrm{NC}_\mathrm{k} = \bigcup_k \mathrm{AC}_\mathrm{k} = \bigcup_k \mathrm{TC}_\mathrm{k} \subseteq P,\end{aligned}$$

where the relation $\mathrm{TC}_\mathrm{i} \subseteq \mathrm{NC}_{\mathrm{i}+1}$ follows from the fact that one can simply replace the unbounded fan-in gates by binary trees of binary gates, in which case the depth growth only by a log factor. For more details on circuit complexity and the notion of uniformity we refer to [26, 32].

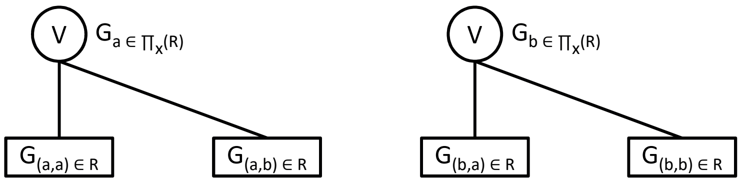
2.4.1 Circuit complexity of Relational Algebra

Since we are interested in proving that the incrementalization of NRC^+ is in a lower complexity class than NRC^+ itself (Section 3.4), we begin by highlighting the sources of "complexity" in NRC^+. To do so, we first discuss the parallel complexity of relational algebra operators as it

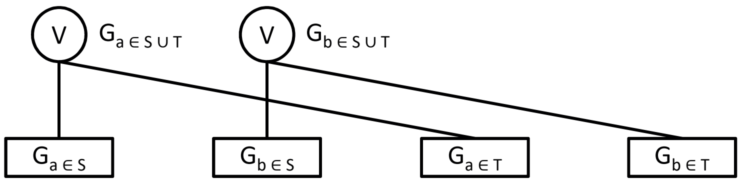
[1] Named after Nicholas (Nick) John Pippenger.


∨
$G_{(a,a)\in\sigma_{x\neq y}(R)}$
$G_{(a,b)\in\sigma_{x\neq y}(R)}$
$G_{(b,a)\in\sigma_{x\neq y}(R)}$
$G_{(b,b)\in\sigma_{x\neq y}(R)}$
$G_{(a,a)\in R}$
$G_{(a,b)\in R}$
$G_{(b,a)\in R}$
$G_{(b,b)\in R}$

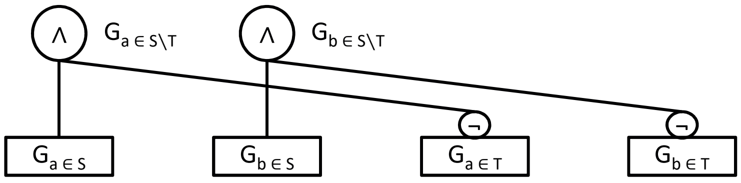
(a) A circuit selecting tuples from relation $R(x, y)$ with distinct components, i.e. $\sigma_{x\neq y}(R)$.



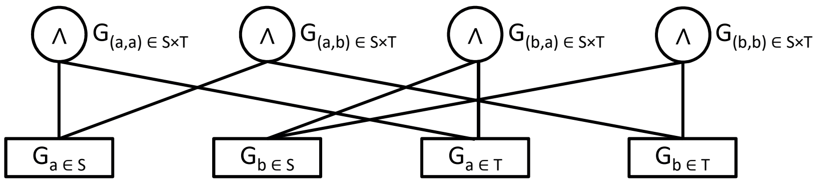
(b) A circuit projecting away the second column of relation $R(x, y)$, i.e. $\Pi_x(R)$.



(c) A circuit unioning relations $S(x)$ and $T(y)$, i.e. $S\cup T$.



(d) A circuit computing the difference between relations $S(x)$ and $T(y)$, i.e. $S\setminus T$.



(e) A circuit producing the cartesian product between relations $S(x)$ and $T(y)$, i.e. $S\times T$.

Figure 2.4 – Circuit implementations of Relational Algebra operators over relations $R(x, y), S(x)$ and $T(y)$, when the active domain consists of two values, a and b.

informs the characterization of the nested relational calculus that we use, considering the close relation between the two. To simplify the presentation we initially assume set semantics, but we explore the implications of bag semantics later on. Throughout the thesis we only discuss data-complexity where the query is considered fixed while only the database is seen as part of the input.

For the selection operator σ_p we have as many output gates as input ones, and each output gate is connected to the corresponding input gate only if the tuple it encodes satisfies the selection predicate. Figure 2.4a presents an example of a circuit which selects the tuples from a binary relation that have distinct components, when the active domain consists of two values, a and b. In its description we leverage the fact that an or-gate with no inputs produces false.

For a particular gate (value) in its output, the circuit implementing the projection operator or-s together all the input gates corresponding to tuples that have that value as their projected column, i.e. $G_{a \in \Pi_x(R)} = \bigvee_{y \in \mathcal{D}} G_{(a,y) \in R}$, where R is a binary relation and $\mathcal{D}$ represents the active domain (see example in Figure 2.4b).

The circuits corresponding to the union and difference operators both match the two gates of their input relations encoding a particular tuple, with the former or-ing them, while the latter and-s them after negating the one belonging to the second argument (this fits the expected semantics as a tuple belongs to $S \setminus T$ if it appears in S but not in T). We illustrate their implementations in Figures 2.4c and 2.4d.

Finally, the circuit implementing the Cartesian product operator and-s together for every pair in its output the two gates from the input relations corresponding to each component (as in Figure 2.4e).

From the description above we can see that most of the relational operators require only the power of NC_0, with the notable exception of the projection operator whose circuit relies on or-gates with unbounded fan-in. This puts relational algebra with set semantics in AC_0.

To see why, lets consider a query that projects away all the columns of an input relation R, producing a boolean result. It is easy to see that the corresponding circuit will have to or-together all the gates of R, and as discussed earlier there are m^{n_f} of them, where n_f is the arity of relation R while m is the size of the active domain. Although n_f is fixed as we assumed the query itself to be fixed, the active domain can be arbitrarily large. By contrast, the fan-in of the other operators of relational algebra is limited by the number of their arguments, which for a fixed query is again fixed.

When bag semantics are considered, the complexity of relational algebra goes to TC_0, since majority gates are needed in order to implement the summing up of multiplicities performed by the projection operator.

If we temporarily set aside the challenges of providing a suitable binary encoding for nested values, which we discuss in Section 3.3.4, one can already asses that our version of nested relational calculus with bag semantics is also in TC_0 as the singleton constructor is a constant-time operation while the semantics of the **flatten** operator is similar to that of the projection operator from relational algebra. In particular, the multiplicity of a tuple in the output of **flatten** is the sum of the multiplicities of that tuple in all the inner bags of the relation being flattened, i.e. given relation $R(X), X : \mathbf{Bag}(A)$:

$$\mathbf{flatten}(R)(a) = \sum_{X:\mathbf{Bag}(A)} X(a) \cdot R(X),$$

where $R(X)$ represents the multiplicity in R of inner bag X, while $X(a)$ denotes tuple a's multiplicity in inner bag X. This summing operation already requires majority-gates of unbounded fan-in, thus placing the nested relational calculus with bag semantics in TC_0, as a lower bound. For a complete proof of the upper bound as well we refer the reader to [34].

3 Deep Incrementalization of Nested Collections

In this chapter 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.

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 NRC$^+ \setminus$ 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 [25], handled it by triggering recomputation of nested bags [45] or defaulted to change propagation [33, 53].

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 [35], incremental processing of NRC$^+$ queries is in a strictly lower complexity class than re-evaluation (NC$_0$ vs. TC$_0$).

The rest of this chapter is organized as follows. We first introduce our approach for the incrementalization of NRC$^+$ queries on a motivating example. The efficient delta processing of a large fragment of NRC$^+$ is discussed in Section 3.2 and in Section 3.3 we show how the full NRC$^+$ can be efficiently maintained.

3.1 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 Section 2.2 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 relation R. We will sometimes simply write Q, if R is obvious from the context.

Example 3. *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* $\mathtt{dir}$. *We define* $\mathtt{related}$ *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)
```

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

M

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

$\mathtt{related}[M]$

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*

$\mathtt{related}[M \uplus \Delta M]$

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 $\mathtt{related}[M]$, give us an expression $\delta(\mathtt{related})[M, \Delta M]$ s.t.:

$$\mathtt{related}[M \uplus \Delta M] = \mathtt{related}[M] \uplus \delta(\mathtt{related})[M, \Delta M].$$

For our example, in order to modify $\mathtt{related}[M]$ into $\mathtt{related}[M \uplus \Delta M]$, without completely replacing the existing tuples, one would have to add the movie Jarhead to the inner bag of related movies for Drive (same genre) and Skyfall (same director). 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. We remark that this situation emerged even though the input was updated via simple bag union.

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

However, our target language of Nested Relational Calculus (NRC^+) 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 [43], 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* [17, 29], 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 (although we investigated this alternative, we found it particularly challenging due to the complex ways in which this custom operation would interact with the existing constructs of the language). 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, along with a series of queries Q^Γ, computing the contents of its inner bags.

3.1.1 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.

$$\begin{aligned}
\texttt{related} &\equiv \textbf{for } m \textbf{ in } M \textbf{ union sng}(\langle m.name, \texttt{relB}(m)\rangle) \\
\texttt{relB}(m) &\equiv \textbf{for } m_2 \textbf{ in } M \textbf{ where } \texttt{isRelated}(m, m_2) \\
&\qquad \textbf{union sng}(m_2.name).
\end{aligned}$$

The translation between Spark and NRC$^+$ is made relatively straightforward by the fact that Spark admits a for-comprehension based syntax, while many of NRC$^+$'s constructs have a direct correspondent in Spark (for e.g. bag union, Cartesian product, **flatten**).

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[2]:

$$\begin{aligned}
\delta(\textbf{for } x \textbf{ in } Q_1 \textbf{ union } Q_2) = \; &\textbf{for } x \textbf{ in } \delta(Q_1) \textbf{ union } Q_2 \\
&\uplus \textbf{for } x \textbf{ in } Q_1 \textbf{ union } \delta(Q_2) \\
&\uplus \textbf{for } x \textbf{ in } \delta(Q_1) \textbf{ union } \delta(Q_2)
\end{aligned} \tag{3.1}$$

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 Section 2.3). 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 is the same problem that we saw with the `related` example above. In Section 3.2 we will show that **sng**(e) is the only construct in our calculus whose efficient incrementalization relies on 'deep' updates.

3.1.2 Maintaining inner bags

In order to facilitate the maintenance of the bags produced by `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

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

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 `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 $\langle \iota, m\rangle$

$$\begin{aligned}\texttt{related}^F &\equiv \textbf{for } m \textbf{ in } M \textbf{ union sng}(\langle m.name, \langle \iota, m\rangle\rangle)\\ \texttt{related}^\Gamma(\ell) &\equiv \textbf{for } m_2 \textbf{ in } M \textbf{ where } \texttt{isRelated}(\ell.2, m_2)\\ &\qquad \textbf{union sng}(m_2.name)\end{aligned}$$

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:

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

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 `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 3.3.3), 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 (3.1) gives us:

$$\begin{aligned}
\delta(\texttt{related}^F) &= \mathbf{for}\ m\ \mathbf{in}\ \Delta M\ \mathbf{union}\ \mathbf{sng}(\langle m.name, \langle \iota, m\rangle\rangle) \\
\delta(\texttt{related}^\Gamma)(\ell) &= \mathbf{for}\ m_2\ \mathbf{in}\ \Delta M\ \mathbf{where}\ \texttt{isRelated}(\ell.2, m_2) \\
&\qquad \mathbf{union}\ \mathbf{sng}(m_2.name)
\end{aligned}$$

We shall prove in Section 3.2 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.

3.1.3 Cost analysis

In the following we show that maintaining `related` incrementally is more efficient than its re-evaluation (for the general case see Section 3.2.2). 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:

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

and by the properties of delta queries and one of the general equivalence laws of the NRC [10], this becomes $V \uplus W$ where

$$V = \textbf{for } r \textbf{ in } \texttt{related}^F[M] \textbf{ union} \quad (3.2)$$
$$\textbf{sng}(\langle r.1, \texttt{related}^\Gamma[M](r.2) \uplus \delta(\texttt{related}^\Gamma)(r.2)\rangle)$$
$$W = \textbf{for } r \textbf{ in } \delta(\texttt{related}^F) \textbf{ union} \quad (3.3)$$
$$\textbf{sng}(\langle r.1, \texttt{related}^\Gamma[M \uplus \Delta M](r.2)\rangle)$$

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 (3.2) the cost of computing V is $O(nd)$ and from (3.3) 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 this approach in detail.

3.2 Incrementalizing IncNRC$^+$

Definition 1. *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. In addition, we define an expression as* input-independent *if it is R-*independent *for all relations R in the database. Similarly, an expression is said to be* input-dependent *if it is R-*dependent *wrt. at least one relation R.*

We define IncNRC$^+$ as the fragment of NRC$^+$ that uses a syntactically restricted singleton construct $\textbf{sng}^*(e)$, where e must be *input-independent* (this restriction impacts only the singleton construct $\textbf{sng}(e)$ that accepts arbitrary expressions e of bag type, whereas the other three singleton constructs in the language $\textbf{sng}(x), \textbf{sng}(\pi_i(x))$ and $\textbf{sng}(\langle\rangle)$ are unaffected, as they are by definition *input-independent*). While this prevents IncNRC$^+$ queries from adding nesting levels to their inputs[3], it does provide the useful guarantee that their deltas do not require deep updates. We take advantage of this fact in this section, as 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 3.3).

[3] We note that the query from Section 3.1 does not belong to IncNRC$^+$.

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

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

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

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

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

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

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

Figure 3.1 – Delta rules for the constructs of 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 [36].

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 3.1, where ΔR is a bag containing the elements to be added/removed from R (with positive/negative multiplicity for insertions/deletions) and we use **let** $X := e_1,\, Y := e_2$ **in** e as a shorthand for **let** $X := e_1$ **in** (**let** $Y := e_2$ **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 3.2.1. *Given an* IncNRC$^+$ *expression* $h[R] : \textbf{Bag}(B)$ *with input* $R : \textbf{Bag}(A)$ *and* *update* $\Delta R : \mathbf{Bag}(A)$*, 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 Appendix A.1.1). □

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

Proof. (sketch) The result follows via structural induction on h and from the definition of $\delta(\cdot)$ (full proof in Appendix A.1.1). □

The lemma above is useful for deriving in a single step the delta of *input-independent* subexpressions (as in Example 4), 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 3.2.1).

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

Example 4. *Taking the delta of the* IncNRC$^+$ *query presented in Example 1 results in:*

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

since $\delta_R(\textbf{for \_ in } p(x) \textbf{ union sng}(x)) = \varnothing$ *(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:* $\text{filter}_p[\Delta R]$.

3.2.1 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 [36], 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 5. *Given bag $R : \mathbf{Bag}(\mathbf{Bag}(A))$ let us consider the first and second order deltas of query h:*

$$\begin{aligned} h[R] &= \mathbf{flatten}(R) \times \mathbf{flatten}(R) \\ \delta(h)[R,\Delta R] &= \mathbf{flatten}(R) \times \mathbf{flatten}(\Delta R) \ \uplus\ \mathbf{flatten}(\Delta R) \times (\mathbf{flatten}(R) \uplus \mathbf{flatten}(\Delta R)) \\ \delta^2(h)[\Delta R,\Delta' R] &= \mathbf{flatten}(\Delta' R) \times \mathbf{flatten}(\Delta R) \ \uplus\ \mathbf{flatten}(\Delta R) \times \mathbf{flatten}(\Delta' R). \end{aligned}$$

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:

$$\begin{aligned} H_0 &= H_0 \uplus H_1[U] \\ H_1[\Delta R] &= H_1[\Delta R] \uplus \delta^2(h)[\Delta R, U]. \end{aligned}$$

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.

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 $h \in \{\Delta R, \mathbf{sng}(x), \mathbf{sng}(\pi_i(x)), \mathbf{sng}^*(e), \emptyset, p, \mathbf{sng}(\langle\rangle)\}$ and:

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

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. The proof follows via structural induction on h and from the definition of $\delta(\cdot)$ and $\deg(\cdot)$. For subexpressions of h which are *input-independent* we use the fact that $\delta(e) = \emptyset$ and $\deg(e) = \deg(\delta(e)) = 0$.

- For $h = R$ we have: $\deg(\delta(R)) = \deg(\Delta R) = 0 = 1 - 1 = \deg(R) - 1$
- For $h = \mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2$ we have the following cases:
  Case 1: $\deg(\delta(e_1)) = \deg(e_1) - 1$ and g is *input-independent*:

  $$\begin{aligned}
  &\deg(\delta(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2)) = \deg(\mathbf{for}\ x\ \mathbf{in}\ \delta(e_1)\ \mathbf{union}\ e_2) =\\
  &= \deg(e_2) + \deg(\delta(e_1)) = \deg(e_2) + \deg(e_1) - 1 = \deg(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2) - 1.
  \end{aligned}$$

  Case 2: $\deg(\delta(e_2)) = \deg(e_2) - 1$ and f is *input-independent*: Analogous to Case 1.

  Case 3: $\deg(\delta(e_2)) = \deg(e_2) - 1$ and $\deg(\delta(e_1)) = \deg(e_1) - 1$:

  $$\begin{aligned}
  &\deg(\delta(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2)) =\\
  &= \deg((\mathbf{for}\ x\ \mathbf{in}\ \delta(e_1)\ \mathbf{union}\ e_2) \uplus (\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ \delta(e_2)) \uplus\\
  &\qquad (\mathbf{for}\ x\ \mathbf{in}\ \delta(e_1)\ \mathbf{union}\ \delta(e_2)))\\
  &= \max(\deg(\mathbf{for}\ x\ \mathbf{in}\ \delta(e_1)\ \mathbf{union}\ e_2), \deg(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ \delta(e_2)),\\
  &\qquad \deg(\mathbf{for}\ x\ \mathbf{in}\ \delta(e_1)\ \mathbf{union}\ \delta(e_2)))\\
  &= \max(\deg(e_2) + \deg(\delta(e_1)), \deg(\delta(e_2)) + \deg(e_1), \deg(\delta(e_2)) + \deg(\delta(e_1)))
  \end{aligned}$$

$$= \max(\deg(e_2) + \deg(e_1) - 1, \deg(e_2) - 1 + \deg(e_1), \deg(e_2) - 1 + \deg(e_1) - 1)$$
$$= \deg(e_2) + \deg(e_1) - 1 = \deg(\textbf{for } x \textbf{ in } e_1 \textbf{ union } e_2) - 1.$$

- For $h = e_1 \times e_2$ the proof is similar to the one for **for** x **in** e_1 **union** e_2 as the definitions of $\delta(h)$ and $\deg(h)$ are similar.
- For $h = e_1 \uplus e_2$ we have the following cases:
  Case 1: $\deg(\delta(e_1)) = \deg(e_1) - 1$ and e_2 is *input-independent*:

  $$\deg(\delta(e_1 \uplus e_2)) = \max(\deg(\delta(e_1)), 0) = \deg(\delta(e_1)) = \deg(e_1) - 1 =$$
  $$= \max(\deg(e_1), 0) - 1 = \deg(e_1 \uplus e_2) - 1.$$

  Case 2: $\deg(\delta(e_2)) = \deg(e_2) - 1$ and e_1 is *input-independent*: Analogous to Case 1.

  Case 3: $\deg(\delta(e_1)) = \deg(e_1) - 1$ and $\deg(\delta(e_2)) = \deg(e_2) - 1$:

  $$\deg(\delta(e_1 \uplus e_2)) = \deg(\delta(e_1) \uplus \delta(e_2)) = \max(\deg(\delta(e_1)), \deg(\delta(e_2)))$$
  $$= \max(\deg(e_1) - 1, \deg(e_2) - 1) = \max(\deg(e_1), \deg(e_2)) - 1 = \deg(e_1 \uplus e_2) - 1.$$

- For $h = \ominus(e)$ we have that $\deg(\delta(e)) = \deg(e) - 1$, therefore:

  $$\deg(\delta(\ominus(e))) = \deg(\ominus(\delta(e))) = \deg(\delta(e)) = \deg(e) - 1 = \deg(\ominus(e)) - 1.$$

- For $h = \textbf{flatten}(e)$ the proof is similar to the one for $\ominus(e)$ as the definitions of $\delta(h)$ and $\deg(h)$ are similar.

□

Corollary 3.2.2. *Given an* IncNRC$^+$ *expression h then* $\deg(h)$ *is the minimum natural number k s.t. $\delta^k(h)$ is* input-independent.

Proof. 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 AC$_0$ vs. NC$_0$ complexity separation between nonincremental and incremental evaluation (Theorem 11).

3.2.2 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.

Our cost model is conservative and does not provide tight upper bounds over the execution time of a query. Nonetheless, this notion is sufficient for capturing the fact that taking the delta of a query results in a cheaper 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.

The cost transformation we propose interprets the constructs of IncNRC$^+$ over cost domains A°, inductively defined for every type A as:

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

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[4]. 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:

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

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

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

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

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 define an update ΔR for a nested bag R as *incremental* if $\text{size}(\Delta R) \prec \text{size}(R)$.

Example 6. *The size of bag* $R : \mathbf{Bag}(String \times \mathbf{Bag}(String))$,

$$R = \{\langle Comedy, \{Carnage\}\rangle, \langle Animation, \{Up, Shrek, Cars\}\rangle\}$$

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 3.2, 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 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.

Similarly, the cardinality of the bag produced by $e_1 \times e_2$ can be determined by multiplying the cardinalities of the bags produced by e_1 and e_2, whereas the size of its elements is obtained by pairing the size of the elements in e_1 and e_2.

Our cost model is designed to produce estimates within constant factors. For example, for the cardinality of the result of bag union we take the maximum cardinality of its inputs, as opposed to their sum. Furthermore, we consider a “call-by-name" evaluation strategy, where values are computed only when needed. Therefore, if part of an intermediate result is projected away, we disregard its cost while if the same bag is computed in several places, we consider its cost every time[5].

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* $\Omega(\text{tcost}_{\mathbf{Bag}(B)}(\mathcal{C}[[h]]))$,

[5] This does not influence the cost in asymptotic terms.

$$\mathcal{C}[[R]] = \text{size}(R) \qquad \mathcal{C}[[\textbf{sng}(x)]]_{\gamma^\circ;\varepsilon^\circ} = \{\varepsilon^\circ(x)\}$$

$$\mathcal{C}[[X]]_{\gamma^\circ;\varepsilon^\circ} = \gamma^\circ(X) \qquad \mathcal{C}[[\textbf{sng}(\pi_i(x))]]_{\gamma^\circ;\varepsilon^\circ} = \{\pi_i(\varepsilon^\circ(x))\}$$

$$\mathcal{C}[[p(x)]] = 1_{\textbf{Bag}(1)} \qquad \mathcal{C}[[\textbf{sng}(\langle\rangle)]] = 1_{\textbf{Bag}(1)}$$

$$\mathcal{C}[[\varnothing]] = 1_{\textbf{Bag}(B)} \qquad \mathcal{C}[[\textbf{sng}^*(e)]] = \{\mathcal{C}[[e]]\}$$

$$\mathcal{C}[[\ominus(e)]] = \mathcal{C}[[e]] \qquad \mathcal{C}[[e_1 \uplus e_2]] = \sup(\mathcal{C}[[e_1]], \mathcal{C}[[e_2]])$$

$$\mathcal{C}[[\textbf{let } X := e_1 \textbf{ in } e_2]]_{\gamma^\circ;\varepsilon^\circ} = \mathcal{C}[[e_2]]_{\gamma^\circ[X:=\mathcal{C}[[e_1]]_{\gamma^\circ;\varepsilon^\circ}];\varepsilon^\circ}$$

$$\mathcal{C}[[e_1 \times e_2]] = \mathcal{C}_o[[e_1]] \cdot \mathcal{C}_o[[e_2]]\{\langle \mathcal{C}_i[[e_1]], \mathcal{C}_i[[e_2]]\rangle\}$$

$$\mathcal{C}[[\textbf{flatten}(e)]] = \mathcal{C}_o[[e]] \cdot \mathcal{C}_{oi}[[e]]\{\mathcal{C}_{ii}[[e]]\}$$

$$\mathcal{C}[[\textbf{for } x \textbf{ in } e_1 \textbf{ union } e_2]] = \mathcal{C}_o[[e_1]]_{\gamma^\circ;\varepsilon^\circ} \cdot \mathcal{C}_o[[e_2]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[e_1]]]}\{\mathcal{C}_i[[e_2]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[e_1]]]}\}$$

Figure 3.2 – The cost transformation $\mathcal{C}[[f]]=\mathcal{C}_o[[f]]\{\mathcal{C}_i[[f]]\} : \mathbb{N}^+\{B^\circ\}$ over the constructs of IncNRC$^+$.

where $\text{tcost}_A : A^\circ \to \mathbb{N}$ *is defined as:*

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

Proof. (Sketch) In order to show that h can be computed within $\Omega(\text{tcost}_{\textbf{Bag}(B)}(\mathcal{C}[[h]])) = \Omega(\mathcal{C}_o[[h]] \cdot \text{tcost}_B(\mathcal{C}_i[[h]]))$ 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 $[[\textbf{sng}(e)]]^L_\varepsilon = \beta_{e,\varepsilon}$ 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:

$$\begin{aligned}
\exp_{Base}(x) &= x \\
\exp_{A_1\times A_2}(\langle x_1, x_2\rangle) &= \langle \exp_{A_1}(x_1), \exp_{A_2}(x_2)\rangle \\
\exp_{\mathbf{Bag}(C)}(\beta_{e,\varepsilon}) &= \textbf{for } y \textbf{ in } [[e]]^L_\varepsilon \textbf{ union sng}(\exp_C(y)).
\end{aligned}$$

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 7. *If we apply the cost transformation to the filter query in Example 1 we get:*

$$\mathcal{C}[[\text{filter}_p[R]]] = \mathcal{C}_o[[R]] \cdot (1 \cdot 1)\{\mathcal{C}_i[[R]]\} = \mathcal{C}[[R]]$$

which corresponds to our expectation that the cost of filtering should be proportional to the cost of the input bag R.

Example 8. *If we apply the cost transformation to the* `related`$[M]$ *query in section 3.1.1 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:*

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

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 $\prec_A$ (for the extended proof see Appendix A.1.2). □

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 $\mathcal{C}[[\text{filter}_p[\Delta R]]] \prec \mathcal{C}[[\text{filter}_p[R]]]$.

3.3 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 3.1, an efficient delta rule for $\textbf{sng}(e)$ 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$^+$. More precisely, we recursively replace nested bags by labels, and separately maintain a set of label dictionaries, where we keep track of the bags that each label represents in the original query.

3.3.1 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:

$$\begin{aligned} Base^F &= Base & Base^\Gamma &= 1 \\ (A_1 \times A_2)^F &= A_1^F \times A_2^F & (A_1 \times A_2)^\Gamma &= A_1^\Gamma \times A_2^\Gamma \\ \textbf{Bag}(C)^F &= \mathbb{L} & \textbf{Bag}(C)^\Gamma &= (\mathbb{L} \mapsto \textbf{Bag}(C^F)) \times C^\Gamma \end{aligned}$$

For instance, the flat representation of a bag of type $\textbf{Bag}(C)$ is a label $l : \mathbb{L}$, whereas its context includes a label dictionary $\mathbb{L} \mapsto \textbf{Bag}(C^F)$, mapping l to the flattened contents of the bag. For the moment we focus on the shredding transformation and defer the in-depth discussion of labels, label dictionaries and the operations one can perform on top of them until Section 3.3.2.

We remark that for tuples of base types: $A^F = A$ and A^Γ is a product of unit types, which we denote by 1^*. We will often ignore void contexts, i.e. those of unit type, as they do not contribute to the final result.

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

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

where $\text{sh}^F(h)$ computes the flat representation of the output bag, while the set of queries in $\text{sh}^\Gamma(h)$ define the context, i.e. the dictionaries corresponding to the labels introduced by $\text{sh}^F(h)$. We note that the shredded expressions depend on the shredded input bag $R^F = \text{sh}^F(R)$, $R^\Gamma = \text{sh}^\Gamma(R)$[6], 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 Figure 3.3. 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.

Notation. We often shorthand $\text{sh}^F(h)$ and $\text{sh}^\Gamma(h)$ 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 $\text{inL}_{\iota,\Pi}$, 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:

$$\begin{aligned}
\text{sh}^F(\mathbf{sng}_\iota(e)) : \mathbf{Bag}(\mathbb{L}) &= \text{inL}_{\iota,\Pi}(\varepsilon) \\
\text{sh}^\Gamma(\mathbf{sng}_\iota(e)) : \mathbb{L} \mapsto \mathbf{Bag}(B^F) \times B^\Gamma &= \langle [(\iota, \Pi) \mapsto e^F], e^\Gamma \rangle.
\end{aligned}$$

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. Since our results hold independently from a particular indexing scheme, we do not explore possible alternatives, although they can be found in the literature [17].

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

[6]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.

$$\begin{array}{ll}
\mathrm{sh}^F(R) : \mathbf{Bag}(A^F) & \mathrm{sh}^F(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2) : \mathbf{Bag}(B^F) \\
\mathrm{sh}^F(R) = \mathbf{for}\ r\ \mathbf{in}\ R\ \mathbf{union}\ \mathrm{s}^F_A(r) & \mathrm{sh}^F(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2) = \mathbf{let}\ x^\Gamma := e_1^\Gamma\ \mathbf{in}\ \mathbf{for}\ x^F\ \mathbf{in}\ e_1^F\ \mathbf{union}\ e_2^F \\
\mathrm{sh}^\Gamma(R) : A^\Gamma & \mathrm{sh}^\Gamma(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2) : B^\Gamma \\
\mathrm{sh}^\Gamma(R) = \mathrm{s}^\Gamma_A & \mathrm{sh}^\Gamma(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2) = \mathbf{let}\ x^\Gamma := e_1^\Gamma\ \mathbf{in}\ e_2^\Gamma
\end{array}$$

$$\begin{array}{lll}
\mathrm{sh}^F(\mathbf{sng}(\pi_i(x))) : \mathbf{Bag}(A_i^F) & \mathrm{sh}^F(\mathbf{sng}_\iota(e)) : \mathbf{Bag}(\mathbb{L}) & \mathrm{sh}^F(\mathbf{sng}(\langle\rangle)) : \mathbf{Bag}(1) \\
\mathrm{sh}^F(\mathbf{sng}(\pi_i(x))) = \mathbf{sng}(\pi_i(x^F)) & \mathrm{sh}^F(\mathbf{sng}_\iota(e)) = \mathrm{inL}_{\iota,\Pi}(\varepsilon) & \mathrm{sh}^F(\mathbf{sng}(\langle\rangle)) = \mathbf{sng}(\langle\rangle) \\
\mathrm{sh}^\Gamma(\mathbf{sng}(\pi_i(x))) : A_i^\Gamma & \mathrm{sh}^\Gamma(\mathbf{sng}_\iota(e)) : (\mathbb{L} \to \mathbf{Bag}(B^F)) \times B^\Gamma & \mathrm{sh}^\Gamma(\mathbf{sng}(\langle\rangle)) : 1 \\
\mathrm{sh}^\Gamma(\mathbf{sng}(\pi_i(x))) = x^{\Gamma_i} & \mathrm{sh}^\Gamma(\mathbf{sng}_\iota(e)) = \langle[(\iota,\Pi) \mapsto e^F], e^\Gamma\rangle & \mathrm{sh}^\Gamma(\mathbf{sng}(\langle\rangle)) = \langle\rangle
\end{array}$$

$$\begin{array}{lll}
\mathrm{sh}^F(\mathbf{sng}(x)) : \mathbf{Bag}(A^F) & \mathrm{sh}^F(\mathbf{flatten}(e)) : \mathbf{Bag}(B^F) & \mathrm{sh}^F(e_1 \times e_2) : \mathbf{Bag}(A_1^F \times A_2^F) \\
\mathrm{sh}^F(\mathbf{sng}(x)) = \mathbf{sng}(x^F) & \mathrm{sh}^F(\mathbf{flatten}(e)) = \mathbf{for}\ l\ \mathbf{in}\ e^F\ \mathbf{union}\ e^{\Gamma_1}(l) & \mathrm{sh}^F(e_1 \times e_2) = e_1^F \times e_2^F \\
\mathrm{sh}^\Gamma(\mathbf{sng}(x)) : A^\Gamma & \mathrm{sh}^\Gamma(\mathbf{flatten}(e)) : B^\Gamma & \mathrm{sh}^\Gamma(e_1 \times e_2) : A_1^\Gamma \times A_2^\Gamma \\
\mathrm{sh}^\Gamma(\mathbf{sng}(x)) = x^\Gamma & \mathrm{sh}^\Gamma(\mathbf{flatten}(e)) = e^{\Gamma_2} & \mathrm{sh}^\Gamma(e_1 \times e_2) = \langle e_1^\Gamma, e_2^\Gamma\rangle
\end{array}$$

$$\begin{array}{llll}
\mathrm{sh}^F(e_1 \uplus e_2) : \mathbf{Bag}(B^F) & \mathrm{sh}^F(\ominus(e)) : \mathbf{Bag}(B^F) & \mathrm{sh}^F(\varnothing) : \mathbf{Bag}(B^F) & \mathrm{sh}^F(p(x)) : \mathbf{Bag}(1) \\
\mathrm{sh}^F(e_1 \uplus e_2) = e_1^F \uplus e_2^F & \mathrm{sh}^F(\ominus(e)) = \ominus(e^F) & \mathrm{sh}^F(\varnothing) = \varnothing & \mathrm{sh}^F(p(x)) = p(x) \\
\mathrm{sh}^\Gamma(e_1 \uplus e_2) : B^\Gamma & \mathrm{sh}^\Gamma(\ominus(e)) : B^\Gamma & \mathrm{sh}^\Gamma(\varnothing) : B^\Gamma & \mathrm{sh}^\Gamma(p(x)) : 1 \\
\mathrm{sh}^\Gamma(e_1 \uplus e_2) = e_1^\Gamma \cup e_2^\Gamma & \mathrm{sh}^\Gamma(\ominus(e)) = e^\Gamma & \mathrm{sh}^\Gamma(\varnothing) = \varnothing_{B^\Gamma} & \mathrm{sh}^\Gamma(p(x)) = \langle\rangle
\end{array}$$

$$\begin{array}{ll}
\mathrm{sh}^F(\mathbf{let}\ X := e_1\ \mathbf{in} e_2) : \mathbf{Bag}(B^F) & \mathrm{sh}^F(\mathbf{let}\ X := e_1\ \mathbf{in}\ e_2) = \mathbf{let}\ X^F := e_1^F,\ X^\Gamma := e_2^\Gamma\ \mathbf{in}\ e_2^F \\
\mathrm{sh}^\Gamma(\mathbf{let}\ X := e_1\ \mathbf{in} e_2) : B^\Gamma & \mathrm{sh}^\Gamma(\mathbf{let}\ X := e_1\ \mathbf{in}\ e_2) = \mathbf{let}\ X^F := e_1^F,\ X^\Gamma := e_2^\Gamma\ \mathbf{in}\ e_2^\Gamma
\end{array}$$

Figure 3.3 – The shredding transformation, where s^F_A and s^Γ_A are described in Figure 3.4.

$$\begin{array}{ll}
\mathrm{s}^F_{Base} : Base \to \mathbf{Bag}(Base) & \mathrm{s}^F_{Base}(a) = \mathbf{sng}(a) \\
\mathrm{s}^\Gamma_{Base} : 1 & \mathrm{s}^\Gamma_{Base} = \langle\rangle \\
\mathrm{s}^F_{A_1 \times A_2} : (A_1 \times A_2) \to \mathbf{Bag}(A_1^F \times A_2^F) & \mathrm{s}^F_{A_1 \times A_2}(a) = \mathbf{for}\ \langle a_1, a_2\rangle\ \mathbf{in}\ \mathbf{sng}(a)\ \mathbf{union} \\
 & \qquad \mathrm{s}^F_{A_1}(a_1) \times \mathrm{s}^F_{A_2}(a_2) \\
\mathrm{s}^\Gamma_{A_1 \times A_2} : A_1^\Gamma \times A_2^\Gamma & \mathrm{s}^\Gamma_{A_1 \times A_2} = \langle \mathrm{s}^\Gamma_{A_1}, \mathrm{s}^\Gamma_{A_2}\rangle \\
\mathrm{s}^F_{\mathbf{Bag}(C)} : \mathbf{Bag}(C) \to \mathbf{Bag}(\mathbb{L}) & \mathrm{s}^F_{\mathbf{Bag}(C)}(v) = \mathcal{D}_C(v) \\
\mathrm{s}^\Gamma_{\mathbf{Bag}(C)} : (\mathbb{L} \mapsto \mathbf{Bag}(C^F)) \times C^\Gamma & \mathrm{s}^\Gamma_{\mathbf{Bag}(C)} = \langle \mathbf{for}\ l\ \mathbf{in}\ \mathrm{supp}(\mathcal{D}_C^{-1})\ \mathbf{union} \\
 & \qquad [l \mapsto \mathbf{for}\ c\ \mathbf{in}\ \mathcal{D}_C^{-1}(l)\ \mathbf{union}\ \mathrm{s}^F_C(c)], \mathrm{s}^\Gamma_C\rangle
\end{array}$$

Figure 3.4 – Shredding nested values: $\mathrm{s}^F_A : A \to \mathbf{Bag}(A^F), \mathrm{s}^\Gamma_A : A^\Gamma$

context $e^\Gamma : \mathbb{L} \mapsto \mathbf{Bag}(B^F) \times B^\Gamma$:

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

$$\begin{aligned}
&\mathrm{u}_{Base}[\langle\rangle] : Base \to \mathbf{Bag}(Base) && \mathrm{u}_{Base}[\langle\rangle](a^F) = \mathbf{sng}(a^F)\\
&\mathrm{u}_{A_1\times A_2}[a^\Gamma] : A_1^F \times A_2^F \to \mathbf{Bag}(A_1 \times A_2) && \mathrm{u}_{A_1\times A_2}[a^\Gamma](a^F) = \mathbf{for}\ \langle a_1^F, a_2^F\rangle\ \mathbf{in\ sng}(a^F)\ \mathbf{union}\\
& && \qquad \mathrm{u}_{A_1}[a^{\Gamma_1}](a_1^F) \times \mathrm{u}_{A_2}[a^{\Gamma_2}](a_2^F)\\
&\mathrm{u}_{\mathbf{Bag}(C)}[a^\Gamma] : \mathbb{L} \to \mathbf{Bag}(\mathbf{Bag}(C)) && \mathrm{u}_{\mathbf{Bag}(C)}[a^\Gamma](l) = \mathbf{sng}(\mathbf{for}\ c^F\ \mathbf{in}\ a^{\Gamma_1}(l)\ \mathbf{union}\\
& && \qquad \mathrm{u}_C[a^{\Gamma_2}](c^F))
\end{aligned}$$

Figure 3.5 – Nesting shredded values: $\mathrm{u}_A[a^\Gamma] : A^F \to \mathbf{Bag}(A)$

where we denote by $e^{\Gamma_1}/e^{\Gamma_2}$ 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:

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

To complete the shredding transformation we also inductively define $\mathrm{s}_A^F : A \to \mathbf{Bag}(A^F)$ and $\mathrm{s}_A^\Gamma : A^\Gamma$, 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:

$$\begin{aligned}
&R^F = \mathbf{for}\ r\ \mathbf{in}\ R\ \mathbf{union}\ \mathrm{s}_A^F(r) \qquad\qquad R^\Gamma = \mathrm{s}_A^\Gamma\\
&\quad R = \mathbf{for}\ r^F\ \mathbf{in}\ R^F\ \mathbf{union}\ \mathrm{u}_A[R^\Gamma](r^F),
\end{aligned}$$

where $\mathrm{s}_A^F, \mathrm{s}_A^\Gamma$ and u_A are presented in Figures 3.4 and 3.5.

Shredding primitive values leaves them unchanged and produces no dictionary ($Base^\Gamma = 1$), while tuples get shredded and nested back component-wise. 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 mappings $\mathcal{D}_C$ and $\mathcal{D}_C^{-1}$:

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

where $\mathcal{D}_C$ should be seen as a function with side effects, that generates different labels for different instances of the same bag 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. 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 **for** x **in** R^F **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 3.5). We note that the definitions themselves also have to be recursively turned to nested form, which is done in $\mathrm{u}_{\mathbf{Bag}(C)}$.

3.3.2 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.

We define a label $l : \mathbb{L}, \mathbb{L} := \mathit{Integer} \times \mathrm{Any}$, to be the pairing between a static index $\iota : \mathit{Integer}$ and a dynamic context $\varepsilon : \mathrm{Any}$, where Any stands for all the possible types of NRC^+_l. The static index is used to distinguish between nested bags created by different instances of the singleton constructor $\mathbf{sng}(e)$, while the dynamic context / value assignment for e's free variables ε identifies a particular bag, from all the possible bags computed by e. Allowing the dynamic context to have any type makes it possible to collect in the same bag labels created in different contexts.

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 9. *Given* $\mathtt{relB}(m) : \mathbf{Bag}(\mathit{String})$*, the query from the motivating example in section 3.1, dictionary* $d = [(\iota, \mathit{Movie}) \mapsto \mathtt{relB}(m)]$ *of type* $\mathbb{L} \mapsto \mathbf{Bag}(\mathit{String})$ *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* $\mathtt{related}^F$*.*

Since $\mathrm{inL}_{\iota,\mathit{Movie}}(m) = \{\langle \iota, m \rangle\}$*, we can indeed recover* $\mathtt{relB}(m)$ *by evaluating:*

$$\mathbf{for}\ l'\ \mathbf{in}\ \mathrm{inL}_{\iota,\mathit{Movie}}(m)\ \mathbf{union}\ d(l').$$

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 $d_1, d_2 : \mathbb{L} \mapsto \textbf{Bag}(B)$, $\text{supp}(d_1 \cup d_2) = \text{supp}(d_1) \cup \text{supp}(d_2)$ and:

$$
\begin{aligned}
&(d_1 \cup d_2)(l) = d_1(l), \text{ if } l \in \text{supp}(d_1) \setminus \text{supp}(d_2)\\
&(d_1 \cup d_2)(l) = d_2(l), \text{ if } l \in \text{supp}(d_2) \setminus \text{supp}(d_1)\\
&(d_1 \cup d_2)(l) = d_1(l), \text{ if } l \in \text{supp}(d_1) \cap \text{supp}(d_2) \;\&\; d_1(l) = d_2(l)\\
&(d_1 \cup d_2)(l) = \text{error}, \text{ if } l \in \text{supp}(d_1) \cap \text{supp}(d_2) \;\&\; d_1(l) \neq d_2(l)
\end{aligned}
$$

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 as highlighted by the following example.

Example 10. *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).*

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

$$
\begin{aligned}
&[l_1 \mapsto \{b_1\}, l_2 \mapsto \{b_2, b_3\}] \cup [l_2 \mapsto \{b_5\}, l_3 \mapsto \{b_4\}] = \textit{error}\\
&[l_1 \mapsto \{b_1\}, l_2 \mapsto \{b_2, b_3\}] \uplus [l_2 \mapsto \{b_5\}, l_3 \mapsto \{b_4\}] = [l_1 \mapsto \{b_1\}, l_2 \mapsto \{b_2, b_3, b_5\}, l_3 \mapsto \{b_4\}]
\end{aligned}
$$

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.

We also 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 (Appendix A.2). This is especially important for guaranteeing that the union of dictionaries performed by the shredded version of bag addition cannot change the expansion of any label.

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

$$
\begin{array}{lll}
\delta(\text{inL}_l) = \emptyset & \delta([l \mapsto e]) = [l \mapsto \delta(e)] & \delta(e_1 \cup e_2) = \delta(e_1) \cup \delta(e_2)\\
\deg(\text{inL}_l) = 0 & \deg([l \mapsto e]) = \deg(e) & \deg(e_1 \cup e_2) = \max(\deg(e_1), \deg(e_2))\\
\mathcal{C}[[\text{inL}_l(a)]] = \{1\} & \mathcal{C}[[[l \mapsto e](l')]] = \mathcal{C}[[e]] & \mathcal{C}[[(e_1 \cup e_2)(l)]] = \sup(\mathcal{C}[[e_1(l)]], \mathcal{C}[[e_2(l)]]),
\end{array}
$$

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:*

$$\begin{aligned}&h[R \uplus \Delta R] = h[R] \uplus \delta(h)[R, \Delta R],\\ &\deg(\delta(h)) = \deg(h) - 1 \qquad \textit{and}\\ &\mathrm{tcost}(\mathcal{C}[[\delta(h)]]) < \mathrm{tcost}(\mathcal{C}[[h]]).\end{aligned}$$

Proof. (sketch) The proof follows immediately via structural induction on h and from the semantics of IncNRC^+_l constructs (for the extended proof see Appendix A.3). □

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 3.3.3.

3.3.3 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:

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

and converting the output back to nested form produces the same result as $h[R]$, that is:

$$\begin{aligned}h[R] = \mathbf{let}\ & R^F := \mathbf{for}\ r\ \mathbf{in}\ R\ \mathbf{union}\ \mathrm{s}^F(r),\\ & R^\Gamma := \mathrm{s}^\Gamma\ \mathbf{in}\\ & \mathbf{for}\ x^F\ \mathbf{in}\ h^F\ \mathbf{union}\ \mathrm{u}[h^\Gamma](x^F),\end{aligned} \tag{3.4}$$

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

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

Proof. We do a case by case analysis on the type being shredded:

- $A = Base$: $\mathbf{for}\ a^F\ \mathbf{in}\ \mathrm{s}^F_{Base}(a)\ \mathbf{union}\ \mathrm{u}_{Base}[\langle\rangle](a^F) = \mathbf{for}\ a^F\ \mathbf{in}\ \mathbf{sng}(a)\ \mathbf{union}\ \mathbf{sng}(a^F) = \mathbf{sng}(a)$

- $A = A_1 \times A_2$

$$\begin{aligned}
&\mathbf{for}\ a^F\ \mathbf{in}\ \mathrm{s}^F_{A_1\times A_2}(a)\ \mathbf{union}\ \mathrm{u}_{A_1\times A_2}[\mathrm{s}^\Gamma_{A_1\times A_2}](a^F) = \\
&= \mathbf{for}\ a^F\ \mathbf{in}\ (\mathbf{for}\ \langle a_1, a_2\rangle\ \mathbf{in}\ \mathbf{sng}(a)\ \mathbf{union}\ \mathrm{s}^F_{A_1}(a_1) \times \mathrm{s}^F_{A_2}(a_2))\ \mathbf{union} \\
&\quad (\mathbf{for}\ \langle a_1^F, a_2^F\rangle\ \mathbf{in}\ \mathbf{sng}(a^F)\ \mathbf{union}\ \mathrm{u}_{A_1}[a_1^\Gamma](a_1^F) \times \mathrm{u}_{A_2}[a_2^\Gamma](a_2^F)) \\
&= \mathbf{for}\ \langle a_1, a_2\rangle\ \mathbf{in}\ \mathbf{sng}(a)\ \mathbf{union}\ \mathbf{for}\ \langle a_1^F, a_2^F\rangle\ \mathbf{in}\ \mathrm{s}^F_{A_1}(a_1) \times \mathrm{s}^F_{A_2}(a_2)\ \mathbf{union} \\
&\quad \mathrm{u}_{A_1}[a_1^\Gamma](a_1^F) \times \mathrm{u}_{A_2}[a_2^\Gamma](a_2^F) \\
&= \mathbf{for}\ \langle a_1, a_2\rangle\ \mathbf{in}\ \mathbf{sng}(a)\ \mathbf{union} \\
&\quad (\mathbf{for}\ a_1^F\ \mathbf{in}\ \mathrm{s}^F_{A_1}(a_1)\ \mathbf{union}\ \mathrm{u}_{A_1}[a_1^\Gamma](a_1^F)) \times (\mathbf{for}\ a_2^F\ \mathbf{in}\ \mathrm{s}^F_{A_2}(a_2)\ \mathbf{union}\ \mathrm{u}_{A_2}[a_2^\Gamma](a_2^F)) \\
&= \mathbf{for}\ \langle a_1, a_2\rangle\ \mathbf{in}\ \mathbf{sng}(a)\ \mathbf{union}\ (\mathbf{sng}(a_1) \times \mathbf{sng}(a_2)) = \mathbf{sng}(a)
\end{aligned}$$

- $A = \mathbf{Bag}(C)$

$$\begin{aligned}
&\mathbf{for}\ l\ \mathbf{in}\ \mathrm{s}^F_{\mathbf{Bag}(C)}(a)\ \mathbf{union}\ \mathrm{u}_{\mathbf{Bag}(C)}[\mathrm{s}^{\Gamma_1}_{\mathbf{Bag}(C)}, \mathrm{s}^{\Gamma_2}_{\mathbf{Bag}(C)}](l) = \\
&= \mathbf{for}\ l\ \mathbf{in}\ \mathcal{D}_C(a)\ \mathbf{union}\ \mathbf{sng}(\mathbf{for}\ c^F\ \mathbf{in}\ \mathrm{s}^{\Gamma_1}_{\mathbf{Bag}(C)}(l)\ \mathbf{union}\ \mathrm{u}_C[\mathrm{s}^{\Gamma_2}_{\mathbf{Bag}(C)}](c^F)) \\
&= \mathbf{for}\ l\ \mathbf{in}\ \mathcal{D}_C(a)\ \mathbf{union} \\
&\quad \mathbf{sng}(\mathbf{for}\ c^F\ \mathbf{in}\ (\mathbf{for}\ c\ \mathbf{in}\ \mathcal{D}_C^{-1}(l)\ \mathbf{union}\ \mathrm{s}^F_C(c))\ \mathbf{union}\ \mathrm{u}_C[\mathrm{s}^\Gamma_C](c^F)) \\
&= \mathbf{for}\ l\ \mathbf{in}\ \mathcal{D}_C(a)\ \mathbf{union} \\
&\quad \mathbf{sng}(\mathbf{for}\ c\ \mathbf{in}\ \mathcal{D}_C^{-1}(l)\ \mathbf{union}\ \mathbf{for}\ c^F\ \mathbf{in}\ \mathrm{s}^F_C(c)\ \mathbf{union}\ \mathrm{u}_C[\mathrm{s}^\Gamma_C](c^F)) \\
&= \mathbf{for}\ l\ \mathbf{in}\ \mathcal{D}_C(a)\ \mathbf{union}\ \mathbf{sng}(\mathbf{for}\ c\ \mathbf{in}\ \mathcal{D}_C^{-1}(l)\ \mathbf{union}\ \mathbf{sng}(c)) \\
&= \mathbf{for}\ l\ \mathbf{in}\ \mathcal{D}_C(a)\ \mathbf{union}\ \mathbf{sng}(\mathcal{D}_C^{-1}(l)) = \mathbf{sng}(a)
\end{aligned}$$

□

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

$$\begin{aligned}
&\mathbf{let}\ R := \mathbf{for}\ r^F\ \mathbf{in}\ R^F\ \mathbf{union}\ \mathrm{u}[R^\Gamma](r^F)\ \mathbf{in}\ h[R] \\
&= \ \mathbf{for}\ x^F\ \mathbf{in}\ h^F\ \mathbf{union}\ \mathrm{u}[h^\Gamma](x^F).
\end{aligned}$$

Proof. The proof consists of a case by case analysis on the structure of h. We detail for $h \in \{\mathbf{sng}(e), \mathbf{flatten}(e)\}$, as the rest of the cases follow in a similar fashion.

- $h = \mathbf{sng}(e)$

$$\begin{aligned}
&\mathbf{let}\ R := \mathbf{for}\ r^F\ \mathbf{in}\ R^F\ \mathbf{union}\ \mathrm{u}[R^\Gamma](r^F)\ \mathbf{in}\ \mathbf{sng}(e)\\
&= \mathbf{sng}(\mathbf{let}\ R := \mathbf{for}\ r^F\ \mathbf{in}\ R^F\ \mathbf{union}\ \mathrm{u}[R^\Gamma](r^F)\ \mathbf{in}\ e)\\
&= \mathbf{sng}(\mathbf{for}\ x^F\ \mathbf{in}\ e^F\ \mathbf{union}\ \mathrm{u}_B[e^\Gamma](x^F))\\
&= \mathbf{sng}(\mathbf{for}\ x^F\ \mathbf{in}\ (\mathbf{for}\ l\ \mathbf{in}\ \mathrm{inL}_{\iota,A^F}(a^F)\ \mathbf{union}\ [(\iota,A^F)\mapsto e^F](l))\ \mathbf{union}\ \mathrm{u}_B[e^\Gamma](x^F))\\
&= \mathbf{for}\ l\ \mathbf{in}\ \mathrm{inL}_{\iota,A^F}(a^F)\ \mathbf{union}\ \mathbf{sng}(\mathbf{for}\ x^F\ \mathbf{in}\ [(\iota,A^F)\mapsto e^F](l)\ \mathbf{union}\ \mathrm{u}_B[e^\Gamma](x^F))\\
&= \mathbf{for}\ l\ \mathbf{in}\ \mathrm{inL}_{\iota,A^F}(a^F)\ \mathbf{union}\ \mathrm{u}_{\mathbf{Bag}(B)}[[(\iota,A^F)\mapsto e^F],e^\Gamma](l)\\
&= \mathbf{for}\ l\ \mathbf{in}\ \mathrm{sh}^F(\mathbf{sng}(e))\ \mathbf{union}\ \mathrm{u}_{\mathbf{Bag}(B)}[\mathrm{sh}^\Gamma(\mathbf{sng}(e))](l)
\end{aligned}$$

- $h = \mathbf{flatten}(e)$

$$\begin{aligned}
&\mathbf{let}\ R := \mathbf{for}\ r^F\ \mathbf{in}\ R^F\ \mathbf{union}\ \mathrm{u}[R^\Gamma](r^F)\ \mathbf{in}\ \mathbf{flatten}(e)\\
&= \mathbf{flatten}(\mathbf{let}\ R := \mathbf{for}\ r^F\ \mathbf{in}\ R^F\ \mathbf{union}\ \mathrm{u}[R^\Gamma](r^F)\ \mathbf{in}\ e)\\
&= \mathbf{flatten}(\mathbf{for}\ l\ \mathbf{in}\ e^F\ \mathbf{union}\ \mathrm{u}_{\mathbf{Bag}(B)}[e^\Gamma](l))\\
&= \mathbf{flatten}(\mathbf{for}\ l\ \mathbf{in}\ e^F\ \mathbf{union}\ \mathbf{sng}(\mathbf{for}\ x^F\ \mathbf{in}\ e^{\Gamma_1}(l)\ \mathbf{union}\ \mathrm{u}_B[e^{\Gamma_2}](x^F)))\\
&= \mathbf{for}\ l\ \mathbf{in}\ e^F\ \mathbf{union}\ (\mathbf{for}\ x^F\ \mathbf{in}\ e^{\Gamma_1}(l)\ \mathbf{union}\ \mathrm{u}_B[e^{\Gamma_2}](x^F))\\
&= \mathbf{for}\ x^F\ \mathbf{in}\ (\mathbf{for}\ l\ \mathbf{in}\ e^F\ \mathbf{union}\ e^{\Gamma_1}(l))\ \mathbf{union}\ \mathrm{u}_B[e^{\Gamma_2}](x^F)\\
&= \mathbf{for}\ x^F\ \mathbf{in}\ \mathrm{sh}^F(\mathbf{flatten}(e))\ \mathbf{union}\ \mathrm{u}_B[\mathrm{sh}^\Gamma(\mathbf{flatten}(e))](x^F)
\end{aligned}$$

□

Theorem 8. *For any* NRC$^+$ *query expression* $h[R] : \mathbf{Bag}(B)$ *property (3.4) holds.*

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

3.3.4 Complexity of shredding

In this section we show that shredding nested bags can be done in the TC$_0$ parallel complexity class. For additional details on parallel complexity classes and the representations of flat relations when processing them via circuits (under set and bag semantics), we refer the reader back to section 2.4.

When it comes to nested values, the F^{Set} representation discussed earlier 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 of the value by mapping each position p in the string to its corresponding symbol s. To do so we assume a non-fixed alphabet that includes the active domain, i.e. all the possible atomic field values.

Example 11. *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):*

$S(p,s) :=$

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

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 "$\langle$","$\rangle$",",","$\{$","$\}$", 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 9. *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 [6], 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:

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

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: $\mathrm{Elem}_C(i,k,l,j)$, which returns true iff the interval (i,j) contains a value of type C between indices k and l, and $\mathrm{StartsWith}_C(l,j)$ / $\mathrm{EndsWith}_C(i,k)$ 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 $\mathrm{Sh}^F_A(i,j,p,s)$:

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

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:

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

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 comma 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 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:

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

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 [6].

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. □

Theorem 10. *Shredding a nested bag of constant size from N^{Str} representation to a series of flat bags in F^{Bag} representation is in* NC_0.

Proof. By examining the proof of Theorem 9 we notice that in the process of shredding a nested bag the extra power of TC_0 wrt. NC_0 is required only in two situations: a) for the existential/universal quantifiers in the queries producing $S^F(p,s)$ and S^Γ and b) for counting when reestablishing consecutive positions for the symbols in the shredded bags as well as in the conversion of the $S^F(p,s)$ and S^Γ flat bags from F^{Set} to F^{Bag} representation.

However, when working with a constant size input, the existential/universal quantifiers can be replaced by a constant number of disjunctions/conjunctions, and therefore their corresponding circuits no longer need gates with unbounded fan-in. Additionally, one can clearly design a NC_0 circuit for counting over a constant number of bits. We can thus conclude that NC_0 is sufficient for shredding a nested bag of constant size from N^{Str} representation to a collection of flat bags in F^{Bag} representation. □

3.4 Complexity class separation

In terms of data complexity, NRC belongs to TC_0 [34, 64], 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 with 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 "$\langle$", "$\rangle$", ",", "$\{$", "$\}$", 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. By contrast, 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 11. *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 3.2.1, 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 $h_i = \delta_d^{(i)}(h)(d)$, $0 \leq i \leq k$, and $\delta_d^{(0)}(h)(d) = h(d)$) 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.

Finally, in Section 3.3.4 we showed that shredding for the initial materialization of the views itself is in TC$_0$, while the shredding of constant-size updates – the only shredding necessary during IVM – is in NC$_0$.

4 Delta-processing for simply-typed lambda calculi

As querying operators are commonly embedded in functional languages, delta processing must also be compatible with the higher-order features of these languages, i.e. their ability to treat functions as values. Moreover, with the advent of cloud platforms where users can interactively submit and modify queries, and then get charged by the amount of data their queries touch, it becomes highly important that one minimizes the additional amount of data processed for every change performed over the original query. For such cases, one can again employ delta derivation techniques over the query itself, where the change to the query is represented in terms of a functional delta over functional values. This way one can isolate and reuse the part of the query that remains unmodified and only execute what is necessary for updating the existing result.

We address these concerns by proposing a delta transformation for a family of simply typed lambda calculi parameterized by a set of primitive types and operators, where each primitive type has a commutative group structure. Moreover, if the primitive operators have cost-efficient delta rules then we show that every expression in the language is also efficiently incrementalizable, and that Recursive IVM is also applicable for this language. As an example, we can embed NRC in such a calculus with all the possible parametric bag types as primitive types and the constructs of NRC as primitive operators. This constitutes an important step in extending the reach of state-of-the-art static techniques, originally developed for relational query languages, to more powerful programming languages.

Key Insight. The most challenging delta rule for the simply-typed lambda calculus is the one for function application $f@a$ when both the functional value f and its argument a are dynamic: $(f+df)@(a+da) = f@(a+da)+df@(a+da)$. Since we need to break the first term into $f@a+\delta(f)@(a,da)$, it follows that we have to be able to derive deltas for any functional value. Therefore, we internalize the delta transformation as another primitive of the language and have lambda abstraction $\lambda x.e$ build a functional value by placing the given term e into a closure. Then, to evaluate the delta primitive on a functional value we simply apply the delta transformation on its internal term and produce: $\lambda x.\lambda dx.\delta_x(e)$. This delta primitive also has its own delta rule, which applies the delta on the update df, as in: $\delta(f+df) = \delta(f)+\delta(df)$.

$$\mathrm{id}_A : A \to A \qquad \pi_i : A_1{\times}A_2 \to A_i \qquad !_A : A \to 1 \qquad \underline{\mathrm{udef}} : A \to B$$

$$\frac{f : A \to B, g : B \to C}{g \circ f : A \to C} \qquad \frac{f_i : C \to A_i, i = 1,2}{\langle f_1, f_2\rangle : C \to A_1{\times}A_2} \qquad \frac{f : C \times A \to B}{\mathrm{curry}(f) : C \to B^A}$$

$$\underline{0}_{\mathbb{D}} : 1 \to \mathbb{D} \qquad \oplus_{\mathbb{D}} : \mathbb{D}{\times}\mathbb{D} \to \mathbb{D} \qquad \ominus_{\mathbb{D}} : \mathbb{D} \to \mathbb{D} \qquad \mathrm{app} : B^A \times A \to B$$

Figure 4.1 – The constructs of $\mathcal{L}$.

We work with a variant of the simply-typed lambda calculus $\mathcal{L}(\mathbb{D}, \underline{\mathrm{udef}})$, corresponding to the language of cartesian closed categories, extended with a set of primitive types $\mathbb{D}$ and functions $\underline{\mathrm{udef}}$. It has the type system:

$$A, B := 1 \mid \mathbb{D} \mid A \times B \mid B^A,$$

and the operators and combinators presented in Figure 4.1, where B^A represents the set of all functions from A to B; id_A, π_i are the identity and projection operators; $\cdot \circ \cdot$, $\langle \cdot, \cdot \rangle$, $\mathrm{curry}(\cdot)$ are the composition, tupling and currying (lambda abstraction) combinators; app stands for function application, and $!_A$ is the bang operator that returns the unique value of the unit type irrespective of its input.

We chose the categorical formulation of simply typed lambda calculus, as opposed to the classical one using name-binders like lambda abstraction, as it simplifies proofs via equational reasoning. Nonetheless, the two formulations are equivalent and we sketch below the translation between them.

Each term $h : C_1 \times \cdots \times C_n \to A$ in $\mathcal{L}$ has a corresponding term $c_1 : C, \cdots, c_n : C \vdash \bar{h} : A$ in the classical formulation, i.e. using name binders, where $c_i, i = 1..n$, are the variables that appear free in $\bar{h}$. For example, $g \circ f : A \to C$ translates to $a : A \vdash \mathit{let}\ b = \bar{f}\ \mathit{in}\ \bar{g} : C$, $\langle f_1, f_2 \rangle : C \to A_1 \times A_2$ corresponds to $c : C \vdash \langle \bar{f}_1, \bar{f}_2 \rangle : A_1 \times A_2$, while the curry combinator $\mathrm{curry}(f) : C \to B^A$ stands for the lambda abstraction $c : C \vdash (\lambda a : A.\bar{f}) : B^A$ of the term $c : C, a : A \vdash \bar{f} : B$.

The semantics of the constructs of $\mathcal{L}$ are given in Figure 4.2 in terms of the equational axioms that they satisfy.

Notation: We use the following shorthands:

$$\underline{\mathrm{bin}}\langle f_1, f_2\rangle = \underline{\mathrm{bin}} \circ \langle f_1, f_2\rangle \qquad \underline{\mathrm{bin}}\langle f_1, f_2, f_3\rangle = \underline{\mathrm{bin}} \circ \langle f_1, \underline{\mathrm{bin}} \circ \langle f_2, f_3\rangle\rangle \qquad f!_A = f \circ !_A,$$
$$f_1 \times f_2 = \langle f_1 \circ \pi_1, f_2 \circ \pi_2\rangle \qquad \underline{\mathrm{bin}}(f_1 \times f_2) = \underline{\mathrm{bin}} \circ (f_1 \times f_2)$$

where $\underline{\mathrm{bin}}$ is an associative binary operator, and we define a set of auxiliary operators that we

$$
\begin{array}{rcl}
f \circ \mathrm{id}_A = \mathrm{id}_B \circ f = f & \quad & (h \circ g) \circ f = h \circ (g \circ f) \\
f_i = \pi_i \circ \langle f_1, f_2 \rangle & & \langle g_1, g_2 \rangle \circ f = \langle g_1 \circ f, g_2 \circ f \rangle \\
!_B \circ f = !_A & & f = \mathrm{app} \circ (\mathrm{curry}(f) \times \mathrm{id}_A) \\
\oplus_{\mathbb{D}} \circ (\oplus_{\mathbb{D}} \times \mathrm{id}_{\mathbb{D}}) = \oplus_{\mathbb{D}} \circ (\mathrm{id}_{\mathbb{D}} \times \oplus_{\mathbb{D}}) \circ \mathrm{rassoc}_{\times} & & \oplus_{\mathbb{D}} \circ \langle \mathrm{id}_{\mathbb{D}}, \underline{0}_{\mathbb{D}} \circ !_{\mathbb{D}} \rangle = \oplus_{\mathbb{D}} \circ \langle \underline{0}_{\mathbb{D}} \circ !_{\mathbb{D}}, \mathrm{id}_{\mathbb{D}} \rangle = \mathrm{id}_{\mathbb{D}} \\
\oplus_{\mathbb{D}} \circ \langle \mathrm{id}_{\mathbb{D}}, \ominus_{\mathbb{D}} \rangle = \oplus_{\mathbb{D}} \circ \langle \ominus_{\mathbb{D}}, \mathrm{id}_{\mathbb{D}} \rangle = \underline{0}_{\mathbb{D}} \circ !_{\mathbb{D}} & & \oplus_{\mathbb{D}} = \oplus_{\mathbb{D}} \circ \mathrm{sw}_{\times}
\end{array}
$$

where:

$$
\begin{array}{ll}
\mathrm{rassoc} : (A \times B) \times C \to A \times (B \times C) & \quad \mathrm{sw}_{\times} : A \times B \to B \times A \\
\mathrm{rassoc} = \langle \pi_{11}, \langle \pi_{21}, \pi_2 \rangle \rangle & \quad \mathrm{sw}_{\times} = \langle \pi_2, \pi_1 \rangle
\end{array}
$$

Figure 4.2 – The equational theory of $\mathcal{L}$.

will use in the rest of the presentation:

$$
\begin{array}{ll}
\pi_{ij} : (B_{11} \times B_{21}) \times (B_{12} \times B_{22}) \to B_{ij} & \quad \mathrm{repair} : (A \times B) \times (C \times D) \to (A \times C) \times (B \times D) \\
\pi_{ij} = \pi_i \circ \pi_j & \quad \mathrm{repair} = \langle \langle \pi_{11}, \pi_{12} \rangle, \langle \pi_{21}, \pi_{22} \rangle \rangle
\end{array}
$$

The repair operator reshuffles the elements from two input tuples, by grouping the first components of the inputs into the first output tuple, while placing the second elements into the second output.

With the goal of delta-processing in mind we require that each primitive type $\mathbb{D}$ has a commutative group $(\mathbb{D}, \underline{0}_{\mathbb{D}}, \oplus_{\mathbb{D}}, \ominus_{\mathbb{D}})$. We also extend addition over product values in a straightforward way by placing in each component of the output tuple the sum of the corresponding components from the input tuples. Similarly, the sum of two functional values f_1, f_2, produces a function that returns for every possible input v the sum of $f_1(v)$ and $f_2(v)$. We show in Appendix A.4.1 that these definitions do indeed exhibit commutative group structures.

4.1 Deriving δ functions

We propose a transformation taking a $\mathcal{L}$ term $h : A \to B$ to its delta expression $\delta(h) : A \times A \to B$, such that given an input a and its change da, $\delta(h)$ computes the corresponding update for the output of h. The details of the delta transformation are presented in Figure 4.3, where given a term $h : C \times A \to B$, its *partial* deltas $\delta_{C,-}(h)/\delta_{-,A}(h)$ wrt. the first/second argument are defined as:

$$
\begin{array}{l}
\delta_{C,-}(h) : (C \times C) \times A \to B = \delta(f) \circ \langle \langle \pi_{11}, \pi_2 \rangle, \langle \pi_{21}, \underline{0}_A ! \rangle \rangle \\
\delta_{-,A}(h) : C \times (A \times A) \to B = \delta(f) \circ \langle \langle \pi_1, \pi_{12} \rangle, \langle \underline{0}_C !, \pi_{22} \rangle \rangle
\end{array}
$$

where $\pi_{ij} = \pi_i \circ \pi_j$ and $\underline{0}! = \underline{0} \circ !$.

$$\begin{array}{lll}
\delta(\mathrm{id}_A) : A \times A \to A & \delta(g \circ f) : A \times A \to C & \delta(!_A) : A \times A \to 1 \\
\delta(\mathrm{id}_A) = \pi_2 & \delta(g \circ f) = \delta(g) \circ \langle f \circ \pi_1, \delta(f) \rangle & \delta(!_A) = !_A \circ \pi_2 \\
 & & \\
\delta(\pi_i) : (A_1 \times A_2)^2 \to A_i & \delta(\langle f_1, f_2 \rangle) : C \times C \to A_1 \times A_2 & \delta(\underline{\mathrm{udef}}) : A \times A \to B \\
\delta(\pi_i) = \pi_i \circ \pi_2 & \delta(\langle f_1, f_2 \rangle) = \langle \delta(f_1), \delta(f_2) \rangle & \delta(\underline{\mathrm{udef}}) = \underline{\mathrm{udef}_\Delta} \\
 & & \\
\delta(\underline{0_{\mathbb{D}}}) : 1 \times 1 \to \mathbb{D} & \delta(\oplus_{\mathbb{D}}) : (\mathbb{D} \times \mathbb{D})^2 \to \mathbb{D} & \delta(\ominus_{\mathbb{D}}) : \mathbb{D}^2 \to \mathbb{D} \\
\delta(\underline{0_{\mathbb{D}}}) = \underline{0_{\mathbb{D}}} \circ \pi_2 & \delta(\oplus_{\mathbb{D}}) = \oplus_{\mathbb{D}} \circ \pi_2 & \delta(\ominus_{\mathbb{D}}) = \ominus_{\mathbb{D}} \circ \pi_2 \\
 & & \\
\delta(\mathrm{curry}(f)) : C \times C \to B^A & \delta(\mathrm{app}) : (B^A \times A)^2 \to B & \delta(\mathrm{delta_o}) : (B^A)^2 \to B^{A \times A} \\
\delta(\mathrm{curry}(f)) = \mathrm{curry}(\delta_{C,-}(h)) & \delta(\mathrm{app}) = \oplus_B \langle \mathrm{app} \circ ((\mathrm{delta_o} \circ \pi_1) \times \mathrm{id}_{A^2}), & \delta(\mathrm{delta_o}) = \mathrm{delta_o} \circ \pi_2 \\
 & \qquad \mathrm{app} \circ (\pi_2 \times \oplus_A) \rangle \circ \mathrm{repair} &
\end{array}$$

Figure 4.3 – Derivation of deltas for the constructs of $\mathcal{L}$.

While the $\delta(h)$ functions derived consider changes with respect to all inputs of h, in practice, only one of h's arguments may change at a time, and deriving partial deltas is preferable as they have more optimization potential.

For most of the constructs in the language the delta derivations follow immediately from their semantics (or in our case equational axioms). We also assume that the delta expressions $\underline{\mathrm{udef}_\Delta}$ for the primitives of the language are provided. However, the delta derivation for functional application is more challenging because, although app distributes wrt. its first argument:

$$\mathrm{app}(f \oplus df, a \oplus da) = \mathrm{app}(f, a \oplus da) \oplus \mathrm{app}(df, a \oplus da),$$

we still have to express $\mathrm{app}(f, a \oplus da)$ in terms of $\mathrm{app}(f, a)$. This essentially requires deriving the delta for a functional value. Therefore, we internalize the δ transformation as an operator on functional values $\mathrm{delta_o} : B^A \to B^{A \times A}$. Informally, given a functional value $f : B^A$, $\mathrm{delta_o}(h)$ recovers its corresponding term, determines its partial delta wrt. A, and then curries the result back into a functional value $\delta f : B^{A \times A}$. Such an operator can be easily implemented if functional values are represented as closures, pairing a term with the set of inputs that have been assigned a value thus far. For example, given term $h : C \times A \to B$ and value $c : C$, $\mathrm{curry}(h)(c)$ will produce a closure containing the term h along with the assignment of c to its first argument.

As with the rest of the operators in $\mathcal{L}$, we give the formal semantics of $\mathrm{delta_o}$ in terms of the axiom it satisfies:

$$\mathrm{delta_o} \circ \mathrm{curry}(f) = \mathrm{curry}(\delta_{-,A}(f)).$$

In other words, $\mathrm{delta_o}$ applied to the lambda produced by $\mathrm{curry}(f)$ results in a functional value of type $B^{A \times A}$, as obtained by currying the partial derivation of f wrt. its second argument.

Having delta_o we can describe $\delta(\text{app})$, as combining the result of the delta of the initial functional value, $\text{delta}_\text{o}(f)(a, da)$, with the result of evaluating the functional update df, on the updated argument, i.e. $df(a \oplus da)$.

For the curry combinator $\text{curry}(f)$ we note that only the first argument of $f : C \times A \to B$ is exposed to incrementalization. Consequently, for its delta rule we use the currying of the partial derivation of $\delta(f)$ wrt. its first argument. In fact, we can see that while the delta of $\text{curry}(f)$ only derives f wrt. the first argument, the derivation wrt. to the second argument is delayed until function application takes place, and is done by delta_o as part of $\delta(\text{app})$.

We now show that the expressions derived via $\delta(\cdot)$ do indeed produce an output update corresponding to the change applied to the input.

Theorem 12. *(Incrementality) For every $\mathcal{L}$ term $h : A \to B$*

$$h \circ \oplus_A = \oplus_B \langle h \circ \pi_1, \delta(h) \rangle,$$

given that this holds for every primitive in the language.

Proof. (sketch) All the cases, except $h = \text{curry}(f)$ and $h = \text{app}$, follow immediately from the definition of δ. For $h = \text{curry}(f)$ we apply curry^{-1} on both sides and make use of the induction hypothesis on f. For $h = \text{app}$, we prove that:

$$\begin{aligned}&\text{app} \circ \oplus_{B^A \times A}((\text{curry}(f) \times \text{id}_A) \times \text{id}_{B^A \times A}) = \\ &= \oplus_B \langle \text{app} \circ \pi_1, \delta(\text{app}) \rangle \circ ((\text{curry}(f) \times \text{id}_A) \times \text{id}_{B^A \times A})\end{aligned}$$

holds for any $f : C \times A \to B$ for which the induction hypothesis holds. The full proof can be found in appendix A.4.2. $\square$

4.2 **Deriving** cost **functions**

In order to establish whether the derived delta functions provide an advantage over full re-evaluation we abstractly interpret the values and expressions in $\mathcal{L}$ over a cost domain, i.e. we introduce a way of assigning costs and cost terms to the values and expressions in $\mathcal{L}$.

We associate to each primitive value in the input a cost proportional to its size and for every $\underline{\text{udef}}$ we introduce $\underline{\text{udef}}^\circ$, which estimates the cost of computing $\underline{\text{udef}}$'s output based on the cost of its input; in particular if $\underline{\text{udef}}$ is a function between primitive types then $\text{cost}(\underline{\text{udef}})(n) = \text{O}\big(\underline{\text{udef}}^\circ(n)\big)$. The cost of producing primitive values can be extended to product types by defining the cost of a product value to be the tuple of the costs of its components. Similarly, the cost of a functional value is going to be a mapping between input and output costs.

Given language $\mathcal{L}(\mathbb{D}, \underline{\text{udef}})$, Figure 4.4 defines the transformation $\text{cost} : \mathcal{L}(\mathbb{D}, \underline{\text{udef}}) \to \mathcal{L}(\mathbb{N}^+, \underline{\text{udef}}^\circ)$, $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$, where $A^\circ = \text{cost}(A)$ represents the cost domain for values of type A. The cost

$$1^\circ = \{1\} \qquad \mathbb{D}^\circ = \mathbb{N}^+ \qquad (A \times B)^\circ = A^\circ \times B^\circ \qquad (B^A)^\circ = (B^\circ)^{A^\circ}$$

$$\begin{array}{lll}
\text{cost}(\text{id}_A) : A^\circ \to A^\circ & \text{cost}(g \circ f) : A^\circ \to C^\circ & \text{cost}(!_A) : A^\circ \to 1^\circ \\
\text{cost}(\text{id}_A) = \text{id}_{A^\circ} & \text{cost}(g \circ f) = \text{cost}(g) \circ \text{cost}(f) & \text{cost}(!_A) = !_{A^\circ} \\
\text{cost}(\pi_i) : A_1^\circ \times A_2^\circ \to A_i^\circ & \text{cost}(\langle f_1, f_2 \rangle) : C^\circ \to A_1^\circ \times A_2^\circ & \text{cost}(\underline{\text{udef}}) : A^\circ \to B^\circ \\
\text{cost}(\pi_i) = \pi_i & \text{cost}(\langle f_1, f_2 \rangle) = \langle \text{cost}(f_1), \text{cost}(f_2) \rangle & \text{cost}(\underline{\text{udef}}) = \underline{\text{udef}}^\circ \\
\text{cost}(\underline{0}_\mathbb{D}) : 1^\circ \to \mathbb{N}^+ & \text{cost}(\oplus_\mathbb{D}) : (\mathbb{N}^+)^2 \to \mathbb{N}^+ & \text{cost}(\ominus_\mathbb{D}) : \mathbb{N}^+ \to \mathbb{N}^+ \\
\text{cost}(\underline{0}_\mathbb{D}) = \underline{1}_{\mathbb{N}^+} & \text{cost}(\oplus_\mathbb{D}) = \max_{\mathbb{N}^+} & \text{cost}(\ominus_\mathbb{D}) = \text{id}_{\mathbb{N}^+}
\end{array}$$

$$\begin{array}{ll}
\text{cost}(\text{app}) : (B^\circ)^{A^\circ} \times A^\circ \to B^\circ & \text{cost}(\text{curry}(f)) : C^\circ \to (B^\circ)^{A^\circ} \\
\text{cost}(\text{app}) = \text{app} & \text{cost}(\text{curry}(f)) = \text{curry}(\text{cost}(f)).
\end{array}$$

Figure 4.4 – Derivation of cost functions for the constructs of $\mathcal{L}$.

transformation can be used to derive cost functions $\text{cost}(h) : A^\circ \to B^\circ$ that compute the cost of producing the output for any term $h : A \to B$ in $\mathcal{L}(\mathbb{D}, \underline{\text{udef}})$, where A° and B° are the cost domains of its input and output, respectively. The $\text{cost}(h)$ function will compute an evaluation cost for h as if all functions in h are inlined, i.e. the cost of producing an intermediate value will be accounted for every time that value is used. Nonetheless, this estimate is sufficient for our goal of comparing the evaluation cost of $\delta(h)$ wrt. that of h in asymptotic terms. To that end we also introduce an ordering relation over the values of the cost domain.

Definition 2. *For every type A° of $\mathcal{L}(\mathbb{N}^+, \underline{\text{udef}}^\circ)$ we define the partial ordering relation $\prec_A$ as follows:*

$$\begin{array}{ll}
1 \preceq_1 1 = \text{true} & \langle \epsilon_{1A}, \epsilon_{1B} \rangle \prec_{A \times B} \langle \epsilon_{2A}, \epsilon_{2B} \rangle = (\epsilon_{1A} \prec_A \epsilon_{2A}) \textit{ and } (\epsilon_{1B} \prec_B \epsilon_{2B}) \\
\epsilon_1 \prec_\mathbb{D} \epsilon_2 = \epsilon_1 < \epsilon_2 & \epsilon_1 \prec_{B^A} \epsilon_2 = \forall a \in A^\circ . \epsilon_1(a) \prec_B \epsilon_2(a).
\end{array}$$

Definition 3. *For every type A° of $\mathcal{L}(\mathbb{N}^+, \underline{\text{udef}}^\circ)$ we define $\underline{1}_A^\circ : 1^\circ \to A^\circ$ and $\max_A^\circ : A^\circ \times A^\circ \to A^\circ$ as follows:*

$$\begin{array}{ll}
\underline{1}_1^\circ = \text{id}_{1^\circ} & \max_1^\circ = !_{1^\circ \times 1^\circ} \\
\underline{1}_\mathbb{D}^\circ = \underline{1}_{\mathbb{N}^+} & \max_\mathbb{D}^\circ = \max_{\mathbb{N}^+} \\
\underline{1}_{A \times B}^\circ = \langle \underline{1}_A^\circ, \underline{1}_B^\circ \rangle & \max_{A \times B}^\circ = (\max_A^\circ \times \max_B^\circ) \circ \text{repair} \\
\underline{1}_{B^A}^\circ = \text{curry}(\underline{1}_B^\circ !_{1^\circ \times A^\circ}) & \max_{B^A}^\circ = \text{curry}(\max_B^\circ \circ \langle \text{app} \circ (\pi_1 \times \text{id}), \text{app} \circ (\pi_2 \times \text{id}) \rangle).
\end{array}$$

It is easy to show that for every $\epsilon_1, \epsilon_2 \in A^\circ$ if $\epsilon_1 \preceq_A \epsilon_2$, then $\max_A^\circ(\epsilon_1, \epsilon_2) = \epsilon_2$.

We can now extend the cost transformation to the group operations for all the types in $\mathcal{L}(\mathbb{D}, \underline{\text{udef}})$:

$$\text{cost}(\underline{0}_A) = \underline{1}^{\circ}_A \qquad \text{cost}(\oplus_A) = \max^{\circ}_A \qquad \text{cost}(\ominus_A) = \text{id}_{A^{\circ}}\,.$$

Finally, we give our result regarding the efficient incrementalization of any expression in $\mathcal{L}$.

Theorem 13. *If every primitive* $\underline{\text{udef}}$ *is efficiently incrementalizable, then the same property holds for the entire language* $\mathcal{L}(\mathbb{D}, \underline{\text{udef}})$*, where a input-dependent term* $h : A \to B$ *is* efficiently incrementalizable *if* $\forall \epsilon, \epsilon_\Delta \in A^{\circ}$ *s.t.* $\epsilon_\Delta \prec \epsilon$*, then*

$$\text{cost}(\delta(h))(\epsilon, \epsilon_\Delta) \prec \text{cost}(h)(\epsilon).$$

Proof. The result follows by induction on the structure of h but we only sketch the proof for the cases of $\text{curry}(f)$ and app, as the rest of the cases follow immediately from the definition. For $h = \text{curry}(f)$ we apply curry^{-1} on each side of the inequality, while for $h = \text{app}$, we consider ϵ_{B^A} to be of the form $\text{cost}(\text{curry}(f))(\epsilon_C)$, and make use of the induction hypothesis on f, where $f : C \times A \to B$. For the full proof see Appendix A.4.2. □

Therefore, given efficiently incrementalizable primitives, any $\mathcal{L}$ term has a delta of lower cost wrt. re-computation.

4.3 Higher-order deltas

Having established a way to derive delta functions for any expression in $\mathcal{L}(\mathbb{D}, \underline{\text{udef}})$ (Section 4.1), we now also extend the notion of higher-order deltas to this language. In this context, higher-order refers to taking deltas of deltas in the spirit of higher-order derivatives.

We remark that the derived delta functions, $\delta(h) : A \times A \to B$, depend on both the input a, and the update da. As the input is usually much larger than the update, one may expect $\delta(h)$'s evaluation cost to be dominated by it. Therefore, it should be beneficial to partially evaluate $\delta(h)$ wrt. the initial input, even before the update da is available, i.e. compute $\Delta h(a)$, where $\Delta h : A \to B^A$, $\Delta h = \text{curry}(\delta(h))$. Thus, when da finally arrives, we can determine the update to $h(a)$ by evaluating $\Delta h(a)(da)$.

In order to continue benefiting from this partial evaluation for future updates as well, we must maintain $\Delta h(a)$ as the input changes. This can also be done incrementally by deriving a second order delta, $\delta(\Delta h)$, such that we have:

$$\Delta h(a \oplus da) = \Delta h(a) \oplus \delta(\Delta h)(a, da).$$

At this point it should not come as a surprise that we can again partially evaluate and incrementally maintain $\delta(\Delta h)$ wrt. the input, i.e. compute $\Delta^2 h(a)$, where $\Delta^2 h : A \to B^{A^A}$,

$\Delta^2 h = \text{curry}(\delta(\Delta h))$. In fact, we could continue incrementally maintaining $\Delta^i h(a)$ based on $\Delta^{i+1} h(a)$ ad infinitum.

However, as a consequence of Theorem 13 we prove that, for constant updates (i.e. $\epsilon_\Delta = 1^\circ_A$) and a big enough n, the cost of the higher order delta $\Delta^n h$ also becomes constant. At that point it is no longer beneficial to incrementally maintain it, as we could just as easily compute it when needed. This generalizes a similar result proven for higher order deltas of relational queries [35].

Corollary 4.3.1. *For any term $h : A \to B$ in an efficiently incrementalizable language $\mathcal{L}(\mathbb{D}, \underline{\text{udef}})$ and an input value with cost $\epsilon \in A^\circ$, there exists $n \geq 1$ such that:*

$$\text{cost}(\Delta^n h)(\epsilon)(1^\circ_A \cdots 1^\circ_A) = 1^\circ_B,$$

where $\Delta^i h = \text{curry}(\delta(\Delta^{i-1} h)), \Delta^0 h = h$, for $i = 1..n$.

Proof. Considering that

$$\text{cost}(\Delta^i h)(\epsilon)(\epsilon_\Delta) = \text{cost}(\delta(\Delta^{i-1} h))(\epsilon, \epsilon_\Delta),$$

we can repeatedly apply Theorem 13 for $\Delta^i h$, i=1..n, and $\epsilon_\Delta = 1^\circ_A$:

$$\text{cost}(\Delta^i h)(\epsilon)(1^\circ_A \cdots 1^\circ_A)(1^\circ_A) \preceq_B \text{cost}(\Delta^{i-1} h)(\epsilon)(1^\circ_A \cdots 1^\circ_A).$$

As the partial order $\preceq_B$ is strict, except for the bottom element 1°_B, we are guaranteed to reach it after a large enough number of steps. □

From corollary 4.3.1 we conclude that when recursively incrementalizing h we can stop deriving and maintaining higher-order deltas once their cost becomes constant.

5 Deep Scaling for Nested Queries

The vision of map-reduce frameworks like Spark, Hadoop or Scope, has been to provide a programming environment for building highly scalable applications where the programmer is shielded from many of the challenges of distributed computing. Despite considerable progress, it still takes advanced expertise when building such systems, and developers must be cautious of a long list of things to avoid if their applications are to perform well. In particular, current systems offer only limited support when operating on nested data in terms of scaling in the presence of large or skewed inner collections. Although we use Spark to exemplify the issue, implement our approach and conduct experiments, the discussion and contributions are relevant to any large scale data processing system whose query language is an extension of nested relational calculus (eg. Pig Latin [55], Scope [14]).

Current frameworks natively provide only *shallow* scaling, wherein the workload is parallelized only at the granularity of top level records, while the sub-computation responsible for processing an inner collection (no matter how large) will always be performed on a single node. This can easily lead to load imbalance and poor resource utilization in the presence of inner collections with skewed sizes or top-level collections with small cardinalities. Moreover, it is up to the programmer to ensure that inner collections do not grow too large, otherwise their application in the best case ends up spilling to disk and in the worst case crashes completely with an out-of-memory exception. By contrast, we say that a framework supports *deep* scaling if it transparently and evenly distributes the processing of nested collections within its workload.

To illustrate the problems, let us consider the following Spark query, which takes a collection of per machine logs for servers running in a data center and determines their corresponding sets of invalid requests that lead to SLA violations (we recall that `RDD` is Spark's representation of a distributed dataset).

```
case class Packet(req_id: Long, time: Long, payload: String)
case class Server(ip: Int, incoming: Set[Packet], outgoing: Set[Packet])

var Server_RDD: RDD[Server] = ...
var Q: RDD[(Int, Set[String])] =
    for (s <- Server_RDD)
    yield (s.ip,
        for (in_pkt <- s.incoming;
             out_pkt <- s.outgoing
             if in_pkt.req_id == out_pkt.req_id &&
                out_pkt.time - in_pkt.time > SLA &&
                out_pkt.payload.startsWith("INVALID"))
        yield in_pkt.payload)
```

Even though the computation required to answer this query is embarrassingly parallel at the granularity of Server_RDD records (and requires no reshuffling), it is nonetheless possible that some workers do considerable more computation than others, if some servers have historically seen significantly more packets than the rest (i.e. more than the total amount of packets divided by the number of workers). Alternatively, if the incoming/outgoing collections associated with each server are large but Server_RDD has low cardinality (i.e. lower than the number of workers), we will end up under-utilizing the available processing nodes. In both cases, the core problem is that the sub-query computing the inner collections of the output is partitioned across nodes only at the granularity of top-level records.

The same issues can be observed also in NoSQL systems, where the inner query would be expressed as an UDF, and thus not be subject to query optimizations. For instance, the join between the *incoming* and *outgoing* collections in our example would be executed in Apache Pig as a Nested Loop Join, whereas a different join algorithm (eg. hash-join) would likely perform better.

Currently, the only way to overcome these challenges is to *manually* re-write the query, to first flatten the inner collections, then perform the join, and finally regroup the result. However, applying this transformation manually without altering the semantics of the initial query is non-trivial and error-prone, and thus feasible only for relatively small nested queries. Moreover, such transformations should ultimately be the subject of a cost-benefit analysis within the query optimization stage based on additional workload statistics.

Even though for our running example coming up with the required re-writes is relatively easy, developing a flattening / shredding transformation that can operate at the level of an entire query language raises considerable challenges. In our work, we take a type-directed approach which enables us to pinpoint the places where nesting occurs within a query, and thus apply re-writings in a localized, compositional and economical way, with many of the language constructs being minimally impacted by the translation.

The limitations of nested collections processing in terms of deep scaling have their root cause in the fact that the construction of nested collections does not distribute wrt. union, as is the case for the majority of collection operators. For example, `(A union B).map(f)` == `A.map(f)` union `B.map(f)`, while `(A union B).groupByKey()` <> `A.groupByKey()` union

`B.groupByKey()`.

Therefore we address this challenge by proposing SLeNDer[1], a compilation framework that given a query operating on nested collections, decouples (shreds) the sub-computations responsible for producing the inner collections from the top-level query and turns them into standalone queries that do indeed distribute wrt. union. Our framework builds on the shredding transformation introduced in Section 3.3.1, and while several other proposals for shredding exist in the literature [17, 29, 68], we favored this one as it is minimally intrusive wrt. the collection API, such that the resulting programs enjoy the same parallelization and optimization opportunities as the top-level queries, it can handle nested-to-nested queries with generalized multiset semantics (which is also essential for efficient incrementalization), and it limits redundancy by producing unique labels per inner collection as opposed to per tuple.

However, we propose a different formalism which draws a clear distinction between types that have an associated *ring* structure and those that do not. In particular, we leverage the algebraic properties of rings in the way we model collections, i.e. as generalized multisets with multiplicities of ring type, as well as in the way we optimize, parallelize and incrementalize their operators. In addition, this separation allows us to cleanly differentiate between two fundamentally different ways of nesting: regular (or key) vs value (or ring) nesting, where the former has a richer semantics while only the latter enjoys the (performance and optimization) benefits of a ring structure (sec. 5.2.1 discusses them in detail) [2]. The shredding transformation employed by SLeNDer is thus designed to convert between these two kinds of nesting, with the goal of simulating the semantics of key nesting, based on queries that perform value nesting instead.

Evaluating queries in shredded form makes it possible to evenly distribute the processing of inner collections since their corresponding label definitions can be easily partitioned between workers. Moreover, within our transformation the level of shredding can be fine-tuned, meaning that we can dynamically apply it only for those inner bags that are a load balancing liability, i.e. whose size exceeds a threshold. This way we can avoid paying the (indirection) costs associated with shredding for the nested collections that are relatively small.

Besides translating a given nested collection program into a semantically equivalent top-level flat query and a corresponding set of dictionary definitions, SLeNDer employs additional transforms to produce efficient plans for evaluating these queries in a distributed environment and finally generate Spark code. For example, while shredding itself produces dictionaries with infinite domains, SLeNDer establishes their finite domain, i.e. the finite domain of the free variables that appear in their definition. It does so based on the domain of the original queries that generated the dictionaries as well as by leveraging equality predicates and primary-key

[1] Skew-Less Nested Data

[2] We remark that the formalism introduced in 2.1 only supports key nesting.

constraints to unify their free variables against the domain of other relations appearing in their defining query.

As discussed in Chapter 3, shredding also enables the efficient incremental maintenance of nested queries and as before, since the label definitions corresponding to inner collections can be simply updated using bag union, we do not have to design special update primitives for applying deep updates, and then have to wrestle with the limited circumstances in which these primitives admit efficient delta rules wrt. the other querying constructs of the language. Moreover, since shredding introduces a single label definition whenever the same inner collection is computed for several top-level records, we only have to incrementally maintain this one label definition, as opposed to the every copy produced by the original query.

For incrementally processing a nested query wrt. small changes of its inputs, SLeNDer generates a trigger program which performs Recursive IVM [36] over its result (in shredded form), i.e. it materializes and incrementally maintains not only the given query, but also the subexpressions of its deltas that do not depend on the update. By doing so, it avoids re-evaluating those sub-expressions over and over again every time those deltas are applied.

Thus far, we have only discussed the benefits shredding provides in terms of load balancing and incrementalizing the computation of nested collections. Nonetheless, its advantages go well beyond that since shredding opens additional optimization opportunities across nesting levels as the one outlined in sec. 5.1.2 and as exploited by query unnesting techniques [23]. Furthermore, if the queries are maintained in shredded form (as many column-oriented storage layers do anyway [50]), a number of joins are no longer necessary for their evaluation, i.e. those responsible for assembling the final nested result. Even if queries down the pipeline do need to perform these joins, one still gets the opportunity to first push their filters and aggregations down to the shredded collections before the join.

The rest of this chaper is structured as follows: In the next section we outline our techniques on a running example. Section 5.2 presents the formalization of the variant of nested relational calculus at the core of the transformations we propose, while Section 5.3 details our approach to shredding it. We then describe in Section 5.4 the architecture of our our compilation framework and the intermediate representation employed by it, highlighting its wider applicability for DSL compilers. Finally, we present our experimental results across a range of queries in section 5.5.

5.1 Motivating use case

We first illustrate our approach on the example query `Q` presented earlier in the chapter. We show how shredding enables its deep scaling, i.e. the parallelization of its processing wrt. to the inner collections of the input, as well as the incremental maintenance of its result. We then discuss the challenges in generating efficient shredded collection programs. In the following some details of shredding have been omitted in order to improve the presentation.

Shredding turns the input relation `Server_RDD` into a flat top level relation `ServerF_RDD`, where inner collections have been replaced by labels, and a couple of dictionaries `ServerG_In` and `ServerG_Out`, containing the definitions of these labels.

```
case class ServerF(ip: Int, lIn: Label, lOut: Label)

var ServerF_RDD: RDD[ServerF] = ...
var ServerG_In: Dictionary[Label, Set[Packet]] = ...
var ServerG_Out: Dictionary[Label, Set[Packet]] = ...
```

Similarly, for the output we have a top level flat query `QF` that only associates to every server ip a label, whereas `QG` determines their corresponding bags of request payloads that exceeded the `SLA`.

```
var QF = for (sF <- ServerF_RDD)
         yield (sF.ip, Label(sF.lIn, sF.lOut))

var QG: Dictionary[Label, Set[String]] = l =>
    for (Label(lIn,lOut) <- l;
         in_pkt <- ServerG_In(lIn);
         out_pkt <- ServerG_Out(lOut)
         if in_pkt.req_id == out_pkt.req_id &&
            out_pkt.time - in_pkt.time > SLA &&
            out_pkt.payload.startsWith("INVALID"))
    yield in_pkt.payload

var QG_dom = (for (s <- ServerF_RDD)
              yield Label(s.lIn, s.lOut)).distinct
```

The labels produced by `QF` encapsulate the identifying parameters that uniquely determine the contents of the collections they replace, in our case the two labels of the input inner collections being joined.

The definition of `QG` does not capture the domain of label parameter `l`, which is instead specified by `QG_dom`. This separation allows us to further optimize the domain inference of `l` based on additional join predicates between the components of `l` and the fields of relations referenced by `QG`. In addition, depending on the execution runtime (shared vs distributed memory), different materialization strategies can be employed when computing `QG` (ranging from lazy evaluation and memoization to full materialization).

Finally, if the nested version of `Q` is needed at any point, it can be easily recovered from its shredding via:

```
for ((ip, l) <- QF) yield (ip, QG(l))
```

5.1.1 Advantages

The main advantage of shredding in terms of exposing the full parallelization potential of collection programs and thus providing *deep* scaling, lies in the fact that the collection associated with a label can be partitioned across multiple nodes, which would otherwise not be possible for an inner bag. This way the computation of inner collections can also be load balanced across cluster resources. In our example, without shredding, the join between the `incoming` and `outgoing` sets of packets for a particular server would take place on a single node. By contrast, upon shredding the join computation of all the bags of payloads in the output can be evenly distributed across all worker nodes.

Deep incrementalization is also enabled by shredding since in order to modify an inner collection one can simply update via bag union its corresponding label definition. In our example, ingesting a new batch of incoming requests for a particular server can be done by simply adding them to its corresponding label definition in `ServerG_In`. Then, this change can be easily propagated through the definition of `QG` in order to determine the delta update for the output (see the definition of `d_QG` below, where `d_ServerG_In` represents the input update). For this kind of changes only the result of `QG` needs updating, whereas `QF` remains unmodified, which corresponds to a deep update of `Q`'s result. This is in line with our expectation that adding new incoming requests for a particular server should result in only that server's bag of outputs being affected by this update.

```
def d_QG(d_ServerG_In: Dictionary[Label, Set[Packet]]) =
    l => for (Label(lIn,lOut) <- l;
              in_pkt <- d_ServerG_In(lIn);
              out_pkt <- ServerG_Out(lOut)
              if in_pkt.req_id == out_pkt.req_id &&
                 out_pkt.time - in_pkt.time > SLA &&
                 out_pkt.payload.startsWith("INVALID"))
         yield in_pkt.payload

QG += d_QG(d_ServerG_In)
ServerG_In += d_ServerG_In
```

In contrast to nested queries, their shredded versions are also amenable to incremental view maintenance (IVM) by rewriting, where regular query engines can be used for delta processing, as opposed to dedicated systems that perform runtime change propagation. In particular, we are able to leverage Recursive IVM [35], a state-of-the-art algebraic rewriting based approach for the delta processing of flat queries that has been shown to provide massive improvements over the view refresh rates obtained using classical IVM.

Recursive IVM materializes and incrementally maintains not only the given query, but its delta queries as well, in particular their sub-expressions that depend only on the input but not on the update. The same principle is applied recursively for higher-order deltas as long as they still depend on the input relations. This approach is grounded in the fact that for a large class of nested queries, their deltas are guaranteed to be simpler than the original query as measured in terms of their dependency on their inputs. Therefore, after a finite number of steps, we will end up with deltas that no longer depend on the original input, but only on the update, and thus there is nothing left to be gained from materializing them further.

Since Recursive IVM places significant constraints on the kinds of queries it can be applied to, it represents a challenging test for the shredding transformation. In particular, shredding must be compositional and not introduce operators outside Recursive IVM's assumed algebraic framework.

In our example, we can extract the subexpression in `d_QG` that does not depend on the input update `d_ServerG_In` and materialize it as `QG_Out`:

```
var QG_Out: Dictionary[Label, Set[Packet]] = l =>
    for (Label(lIn,lOut) <- l;
         out_pkt <- ServerG_Out(lOut)
         if out_pkt.payload.startsWith("INVALID"))
    yield out_pkt
```

and then re-write the delta query as:

```
def d_QG(d_ServerG_In: Dictionary[Label, Set[Packet]]) =
    l => for (Label(lIn,lOut) <- l;
              in_pkt <- d_ServerG_In(lIn);
              out_pkt <- QG_Out(l)
              if in_pkt.req_id == out_pkt.req_id &&
                 out_pkt.time - in_pkt.time > SLA)
         yield in_pkt.payload
```

This way `QG_Out` is no longer recomputed by `d_QG` for every change applied to `ServerG_In`, but only when its value gets modified due to an evolving `ServerG_Out`. The maintenance of `QG_Out` is handled analogously via its corresponding delta query:

```
def d_QG_Out(d_ServerG_Out: Dictionary[Label, Set[Packet]]) =
    l => for (Label(lIn,lOut) <- l;
              out_pkt <- d_ServerG_Out(lOut)
              if out_pkt.payload.startsWith("INVALID"))
         yield out_pkt
```

The process of recursive incrementalization stops with the delta of `QG_Out` since it does not depend on the input anymore, and thus there is no sub-result to be materialized and reused across its subsequent applications.

5.1.2 Building efficient shredded programs

Thus far we have highlighted the benefits of shredding in its ability to enable deep scaling and recursive incrementalization, while ignoring the issues related to the materialization and partitioning / replication of label definitions. We address these issues next in the context of map-reduce frameworks and we showcase the final query plan that can be achieved this way.

We start by turning the dictionaries into finite domain relations based on the `QG_dom` domain definition, where the `groupByLabel` operator performs only a local grouping of the elements of a label definition. Thus, the bag corresponding to a single label ends up partitioned across several nodes.

```
var QG: Dictionary[Label, Set[String]] = (
    for (sF <- ServerF_RDD;
         in_pkt <- ServerG_In(sF.lIn);
         out_pkt <- ServerG_Out(sF.lOut)
         if in_pkt.req_id == out_pkt.req_id &&
            out_pkt.time - in_pkt.time > SLA &&
            out_pkt.payload.startsWith("INVALID"))
    yield (Label(sF.lIn,sF.lOut), in_pkt.payload) )
    .groupByLabel()
```

We then push filtering to the base relations and introduce a join operator with an extended join key, i.e. dictionary label + `req_id`, which has the potential to alleviate any existing skew in the cardinality of `sF.lIn` / `sF.lOut` label definitions. Thus shredding not only distributes the processing of an inner collection on multiple workers, but it also opens up the opportunity to diminish skewness by leveraging join predicates across nesting levels.

```
var QG_In =
    for (sF <- ServerF_RDD;
         in_pkt <- ServerG_In(sF.lIn))
    yield ((Label(sF.lIn,sF.lOut), in_pkt.req_id), in_pkt)
var QG_Out =
    for (sF <- ServerF_RDD;
         out_pkt <- ServerG_Out(sF.lOut)
         if out_pkt.payload.startsWith("INVALID"))
    yield ((Label(sF.lIn,sF.lOut), out_pkt.req_id), out_pkt)

var QG: Dictionary[Label, Set[String]] = (
    for (((l,_),(in_pkt,out_pkt)) <- QG_In .join(QG_Out)
         if out_pkt.time - in_pkt.time > SLA)
    yield (l, in_pkt.payload) )
    .groupByLabel()
```

If `ServerF_RDD` is known to be relatively small it gets replicated across the cluster such that label lookups into `ServerG_In` and `ServerG_Out` can be performed locally, i.e. a record of `ServerF_RDD` containing label ℓ is collocated with every partition of ℓ's definition. Moreover, these lookups return only the local partition of the definition such that both `QG_In` and `QG_Out` do not require any data shuffling.

In the absence of any statistics the result of the join between `QG_In` and `QG_Out` is randomly spread across workers, resulting in the random partitioning of the label definitions computed by `QG` as well.

Alternatively, if information about the sizes of label definitions is available, we split the labels into heavy vs light "hitters", i.e. those with large vs small definitions. Then, when computing `QG` we make sure that all tuples corresponding to a light hitter end up on the same node and we only partition the processing of heavy hitters. This way the label definitions produced by `QG` will no longer be fragmented all over the cluster if their cardinality is low, leading to a significantly lower cost for constructing the nested version of the result.

Since the shredding transformation we employ supports partial shredding, we can go even further and only shred those inner collections that are big enough that they would cause load imbalance, while keeping the rest in nested form. This optimization trades off more computation when producing the shredded result for less lookup overheads when consuming it, either by further processing or by the final conversion to nested form.

5.2 Nested Ring Calculus

In order to apply the optimizations described in the previous section, SLeNDer works with queries expressed in a version of Nested Relational Calculus (NRC) with generalized multiset semantics, where collections are modeled by mappings from keys to multiplicities $\mathbb{K} \to \mathcal{R}$, and any value with an associated *ring* structure can be used as multiplicity. In this formalism bags are represented as maps between tuples and integers $\mathbb{K} \to \mathit{Int}$, which we typically denote by $\mathit{Bag}(\mathbb{K})$ (we use integers as opposed to only natural numbers in order to also model deletions). This way of representing collections underlines the potential for parallelization and incrementalization for many of NRC's operators whose semantics are defined in terms of ring operations, and allows for a clear separation between the constructs that are amenable to such optimizations (eg. map, flatten) from those that require special consideration (eg. nesting). Moreover, since bags themselves have a ring structure as well, they can also be used as multiplicities in what we call *value* or *ring* nesting. For instance, dictionaries make use of value nesting as they associate labels to their defining bags **Label** $\to \mathit{Bag}(\mathbb{K})$. By contrast, key nesting takes a ring value and places it as part of the key of a mapping.

More concretely, given a bag of strings $X : \mathit{Bag}(String)$, key nesting associates to it a ring value $r : \mathcal{R}$ to produce a mapping $\{X \mapsto r\} : \mathit{Bag}(String) \to \mathcal{R}$ from X to r, whereas, value nesting associates it to a key value $k : \mathbb{K}$ resulting in $\{k \mapsto X\} : \mathbb{K} \to \mathit{Bag}(String)$.

In the context of incrementalization, the ring structure of $\mathbb{K} \to Bag(\mathit{String})$ allows us to apply update dX to the contents of X by unioning $\{k \mapsto X\}$ with $\{k \mapsto dX\}$. However, the same is not possible for the result of key nesting, considering that the ring structure of X becomes inaccessible after being incorporated in $\{X \mapsto r\}$, and thus it no longer has any contribution to outer ring structure of $Bag(\mathit{String}) \to \mathcal{R}$. By applying shredding to key nesting, we essentially decouple the identity aspect of values from their ring structure, i.e. we preserve the ring structure of X as part of a label definition that employs value nesting $\{\ell \mapsto X\} : \textbf{Label} \to Bag(\mathit{String})$, while key nesting itself is applied only to the identifying label, producing mapping $\{\ell \mapsto r\} : \textbf{Label} \to \mathcal{R}$.

In order to manage the distinction between key and value nesting, NRC's type system has a bimodal organization that considers $\mathcal{R}$ types that have an associated ring structure, and thus can play the role of multiplicities, separately from $\mathbb{K}$ types which can only appear as keys:

$$
\begin{aligned}
&\mathcal{R} := \mathit{Int} \mid \mathit{Double} \mid \mathcal{R} \times \mathcal{R} \mid \mathbb{K} \to \mathcal{R} \\
&\mathbb{K} := \texttt{Unit} \mid \texttt{Dom} \mid \mathbb{K} \times \mathbb{K} \mid \textbf{Label} \mid [\mathcal{R}] \\
&\textbf{RKey}(\mathcal{R}) \equiv \textbf{Label} \mid [\mathcal{R}],
\end{aligned}
$$

where tuples are obtained via product types $\cdot \times \cdot$, `Unit` is the type of the empty tuple, and `Dom` represents the active domain of the database.

The boxed type $[\mathcal{R}]$ offers a view of ring types as keys, while recalling their underlying ring structure (that could be exploited either in terms of parallelization or incrementalization). It marks the places where shredding can be applied in order to convert the existing key nesting into value nesting using labels and corresponding label dictionaries. While full shredding replaces $[\mathcal{R}]$ types with **Label** types, partial shredding uses the union type $\textbf{RKey}(\mathcal{R})$ allowing for both nested and shredded values. In particular, $\textbf{RKey}(\mathcal{R})$ acts as a function on ring types, forgetting their ring structure and returning a polymorphic representative key type: either the boxed version of the ring type itself, or a reference **Label**.

Example 12. *In our formalism the packets from the motivating example have type* $\mathtt{Packet} \equiv \mathtt{Long} \times \mathtt{Long} \times \mathtt{String}$*, while the input* `Server_RDD` *has type:*

$$Bag(\mathtt{Int} \times [Bag(\mathtt{Packet})] \times [Bag(\mathtt{Packet})]),$$

where the types of the incoming and outgoing collections of packets have been boxed so they can be used as part of a key type.

Upon full shredding the `ServerF_RDD` *gets type* $Bag(\mathtt{Int} \times \textbf{Label} \times \textbf{Label})$*, while allowing for partial shredding is reflected in its type as*

$$Bag(\mathtt{Int} \times \textbf{RKey}(Bag(\mathtt{Packet})) \times \textbf{RKey}(Bag(\mathtt{Packet}))).$$

The two additional dictionaries `ServerG_In` *and* `ServerG_Out` *both have type* $\textbf{Label} \to Bag(\mathtt{Packet})$.

The **RKey**$(\cdot)$ type constructor comes with casting operators toK$(\cdot)$: **RKey**$(\mathcal{R})$ and fromK$(\cdot)$: $\mathcal{R}$, which turn a ring value $r : \mathcal{R}$ into a **RKey**$(\mathcal{R})$ and vice-versa such that fromK(toK(r)) = r. It is through the semantics of these operators that we control whether shredding is performed and to which degree. In their standard form they simply box and unbox $\mathcal{R}$ values into $[\mathcal{R}]$ values. By contrast, upon shredding toK() returns the label associated with the nested ring value in a corresponding dictionary, whereas fromK() accepts a dictionary as an additional argument and looks up the definition of a given label. Finally, in the case of partial shredding, toK() decides at runtime whether to perform boxing or to produce a label, whereas fromK() does unboxing or label lookup depending on its input.

We present the constructs of our core calculus and then we discuss the semantics of the operators derived from the ring structures in its type system, along with those operating on booleans or performing nesting:

$$\begin{aligned} r &:= c \mid X \mid \langle r_1, r_2 \rangle \mid r.\_i \mid \{x \Rightarrow r\} \mid \text{let } X := r_1 \text{ in } r_2 \\ &\mid r_1 + r_2 \mid -r \mid r_1 * r_2 \mid r_1 \cdot r_2 \mid \text{sum}(r) \\ &\mid p(x) \mid \neg(r) \mid \text{sng}(e, r) \mid \text{fromK}(e, r) \\ e &:= \mathrm{c} \mid x \mid \langle e_1, e_2 \rangle \mid e.\_i \mid \text{toK}(r), \end{aligned}$$

where expressions of ring and key type are denoted by r and e, respectively, with corresponding constants (c and c), as well as tupling $\langle\cdot,\cdot\rangle$ and projection ($.\_i$) operators. Infinite mappings are defined via $\{x \Rightarrow r\}$, where r is a ring expression with a free variable x of key type.

We remark that in contrast to the formalism introduced in 2.1, we only have one singleton construct as opposed to four different ones. This was indeed made possible by the distinction between key and ring types, and their corresponding casting pair toK(r)/fromK(e, r). Moreover, we no longer introduce high-level constructs like for-comprehensions, but we work with their underlying ring-based operators as this fully exposes the potential for optimizations via re-writings. More importantly, the formalism we introduce in this chapter is strictly more expressive than the one presented in Section 2.1, due to the generic way of representing collections as key-value mappings. This representation allows us to uniformly capture a host of collection types, including bags and dictionaries, as opposed to using distinct types for each of them. Furthermore, it enables us to natively capture both key and value nesting, as opposed to only key nesting with the formalism in 2.1, thus providing programmers with the flexibility to choose between the two, depending on their requirements. Finally, we mention that the results presented in Chapter 3 wrt. the efficiency of incrementalization or the ability to perform recursive delta derivation over the positive fragment of the language carry over to the nested ring calculus we introduce in the current chapter as well.

Ring Calculus. Given ring structure $(\mathcal{R}, 0_{\mathcal{R}}, +_{\mathcal{R}}, -_{\mathcal{R}}, 1_{\mathcal{R}}, *_{\mathcal{R}})$, we obtain an analogous ring structure for mapping type $\mathbb{K} \to \mathcal{R}$ by extending element-wise $\mathcal{R}$'s operations over key space $\mathbb{K}$. For example, the union of mappings $X +_{\mathbb{K}\to\mathcal{R}} Y$ associates to every key $k : \mathbb{K}$ in its result: $X(k) +_{\mathcal{R}} Y(k)$. If X and Y are bags, then $X +_{Bag(\mathbb{K})} Y$ produces their bag union.

Moreover, we introduce a product operation that can multiply arguments of different ring types, defined inductively as:

$$\begin{array}{ll}
\text{if } X : \mathit{Int}, Y : \mathit{Int}, & \text{then } X \cdot Y : \mathit{Int} \text{ and } X \cdot Y = X * Y, \\
\text{if } X : \mathbb{K} \to \mathcal{R}, Y : \mathit{Int}, & \text{then } X \cdot Y : \mathbb{K} \to (\mathcal{R} \cdot \mathit{Int}), \text{ and } (X \cdot Y)(k) = X(k) \cdot Y, \text{for all } k : \mathbb{K}, \\
\text{if } X : \mathbb{K}_1 \to \mathcal{R}_1,\ Y : \mathbb{K}_2 \to \mathcal{R}_2, & \text{then } X \cdot Y : (\mathbb{K}_1 \times \mathbb{K}_2) \to (\mathcal{R}_1 \cdot \mathcal{R}_2), \text{ and} \\
 & (X \cdot Y)(\langle k_1, k_2 \rangle) = X(k_1) \cdot Y(k_2), \text{for all } k_i : \mathbb{K}_i, i = 1,2,
\end{array}$$

where $\mathcal{R}_1 \cdot \mathcal{R}_2$ represents the output type of the product between arguments of type $\mathcal{R}_1$ and $\mathcal{R}_2$. Given bags $X : \mathit{Bag}(\mathbb{K}_1)$ and $Y : \mathit{Bag}(\mathbb{K}_2)$, then $X \cdot Y : \mathit{Bag}(\mathbb{K}_1 \times \mathbb{K}_2)$ produces their Cartesian product.

Based on the product defined above we also extend the multiplication operation to accept mappings with different ring types, i.e. given mappings $X : \mathbb{K} \to \mathcal{R}_1$ and $Y : \mathbb{K} \to \mathcal{R}_2$, then $X * Y : \mathbb{K} \to (\mathcal{R}_1 \cdot \mathcal{R}_2)$ and $(X * Y)(k) = X(k) \cdot Y(k)$ for all $k : \mathbb{K}$.

Finally, we leverage the addition operation of rings to introduce a summation construct over mappings $X : \mathbb{K} \to \mathcal{R}$:

$$\text{sum}(X) : \mathcal{R} \qquad\qquad \text{sum}(X) = \sum_{k:\mathbb{K}} X(k).$$

This operator is well-defined only for mappings with a finite number of non-zero valued elements, and when applied to bags it returns their count aggregate. Moreover, we can use it to define the for-comprehension construct for mappings as:

$$\text{for}(x \leftarrow r_1) \text{ collect } r_2 \equiv \text{sum}(r_1 * \{x \Rightarrow r_2\}),$$

where r_1 is a finite domain mapping $\mathbb{K} \to \mathcal{R}_1$, $r_2 : \mathcal{R}_2$ is an expression with a free variable $x : \mathbb{K}$. For every mapping of r_1, it binds its key $k : \mathbb{K}$ to x and multiplies its associated ring value $r_1(k) : \mathcal{R}_1$ with the result of r_2, and finally aggregates every such result to obtain a single ring value of type $\mathcal{R}_1 \cdot \mathcal{R}_2$. More precisely we have:

$$\text{sum}(r_1 * \{x \Rightarrow r_2\}) = \sum_{k:\mathbb{K}_1} r_1(k) \cdot r_2[k/x]$$

Example 13. *Using the constructs of the ring calculus defined thus far, we can already express the flattening of a nested collection $X : \mathit{Bag}(\textbf{RKey}(\mathit{Bag}(\mathbb{K})))$ as*

$$\text{flt}(X) = \text{for}(x \leftarrow X) \text{ collect fromK}(x),$$

where $\text{fromK}(x) : \mathit{Bag}(\mathbb{K})$*, with* $x : \textbf{RKey}(\mathit{Bag}(\mathbb{K}))$*, unpacks the boxed inner bag bound by* x*. Considering that the type of* X *can be expanded to* $\textbf{RKey}(\mathit{Bag}(\mathbb{K})) \to \mathit{Int}$*, it can be seen that* $\text{flt}(X)$ *produces a result of the desired type* $\mathit{Bag}(\mathbb{K})$*. It multiplies each inner bag with its corresponding top level multiplicity, and then unions all the resulting bags via the summation* *construct to obtain a single output bag containing the flattened input.*

Booleans. Our type system does not have a dedicated boolean type, but instead we model boolean values via integers, with false represented by 0, and true by any strictly positive integer. This way, logical "or" and "and" can be performed using integer addition and multiplication, respectively, while negation is introduced via $\neg(\cdot)$, which returns false on non-zero arguments and true (i.e. 1) otherwise. Finally, predicates take as input key values and produce booleans.

Example 14. *Given a bag* $X : Bag(\mathbb{K})$ *and predicate* $p(\cdot) : Int$*, we can express* $\text{exists}_p(X)$ *and* $\text{forall}_p(X)$ *as:*

$$\begin{aligned} \text{exists}_p(X) &= \text{for}(x \leftarrow X) \text{ collect } p(x) \\ \text{forall}_p(X) &= \neg(\text{for}(x \leftarrow X) \text{ collect } \neg p(x)). \end{aligned}$$

5.2.1 Key vs. Value Nesting

In order to capture nesting we use the singleton constructor $\text{sng}(k, r)$ that places a given key $k : \mathbb{K}$ and ring value $m : \mathcal{R}, m \neq 0_{\mathcal{R}}$, as the single non-zero valued element $k \mapsto m$ of a mapping $\mathbb{K} \to \mathcal{R}$. To improve the presentation we will use the following notation:

$$\begin{aligned} &\text{for}(x \leftarrow r_1 \text{ if } p(x)) \text{ yield } e \mapsto r_2 \equiv \\ &\qquad\qquad \equiv \text{for}(x \leftarrow r_1) \text{ collect } \text{sng}(e, p(x) \cdot r_2), \end{aligned}$$

where both the predicate p and the ring expression r_2 are optional, and can default to true and 1, respectively. Moreover we will simply write $\text{sng}(e)$ if the value associated to e is 1.

We showcase the difference between key and value (or ring) nesting on a query that takes $X : Bag(\mathbb{K}_1 \times \mathbb{K}_2)$ as input, and produces for every key $k_1 : \mathbb{K}_1$ in X, its corresponding group of keys $k_2 : \mathbb{K}_2$ satisfying predicate p.

Answering this query using value nesting returns a mapping $\mathbb{K}_1 \to Bag(\mathbb{K}_2)$ as defined by:

$$\text{nestR} = \text{for}(\langle k_1, k_2 \rangle \leftarrow X \text{ if } p(k_2)) \text{ yield } k_1 \mapsto \text{sng}(k_2)$$

By contrast, key nesting is achieved by passing a boxed collection as part of the key argument of yield and produces a bag of type $Bag(\mathbb{K}_1 \times \textbf{RKey}(Bag(\mathbb{K}_2)))$:

$$\begin{aligned} \text{nestK} &= \text{for}(\langle k_1, k_2 \rangle \leftarrow X) \text{ yield } \langle k_1, \text{toK}(\text{group}(k_1)) \rangle \\ \text{group}(k) &= \text{for}(\langle k_1, k_2 \rangle \leftarrow X \text{ if } k == k_1 \,\&\&\, p(k_2)) \text{ yield } k_2 \end{aligned}$$

where $k == k_1$ is a predicate testing for equality between k and k_1.

Even though key and value nesting on the surface achieve similar results, their semantics make them quite different and from a performance perspective, we remark that value nesting takes linear-time in the size of its input, whereas key nesting is quadratic. We further illustrate the distinction on the following example.

Example 15. *Given input bag*

$$X = \{\langle a,1\rangle \mapsto 4, \langle b,2\rangle \mapsto 3, \langle b,3\rangle \mapsto 2, \langle b,4\rangle \mapsto 1\},$$

and considering predicate $p(k_2) = k_2 > 2$*, then value nesting produces bag* $Y_R = \{b \mapsto \{3 \mapsto 2, 4 \mapsto 1\}\}$*, while key nesting results in:* $Y_K = \{\langle a, \{\}\rangle \mapsto 4, \langle b, \{3 \mapsto 2, 4 \mapsto 1\}\rangle \mapsto 6\}$.

From the above we can see that only key nesting preserves the top level multiplicities of tuples and thus can properly keep track of keys whose corresponding groups are empty (as is the case for $k_1 = a$*). Moreover, while one can leverage the ring structure of* Y_R *to partition it into* $\{b \mapsto \{3 \mapsto 2\}\}$ *and* $\{b \mapsto \{4 \mapsto 1\}\}$*, the result of* Y_K *cannot be distributed in a similar way since one cannot simply partition the keys of a mapping without completely altering its semantics.*

5.2.2 Delta Derivation

When updating the results of queries wrt. input changes, we leverage the ring structures and rich algebraic framework associated to the types in our calculus in order to apply the changes as well as to derive delta expressions $\delta_X(r)$ that compute their value.

Updates are applied via the addition operation of the input's ring type (for eg. bag union in the case of bags), while the removal of a mapping $k \mapsto m$ from a collection is modeled by adding a new mapping associating k to the inverse of its current multiplicity $-m$ (this results in a final multiplicity of 0, meaning k is no longer part of the collection).

We leverage the distinction between key and ring types at the level of our type system to quickly identify those constructs in our language that are amenable to (efficient) delta processing and we present their delta rules in figure 5.1.

While for most operators we can exploit their linearity or distributivity wrt. addition in order to limit the work performed by their delta expression, this is not the case for negation, constructing singletons or extracting ring values from key types, where we have to default to recomputation.

Nonetheless, in the case of $\text{sng}(e,r)$ and $\text{fromK}(e,r)$ we notice that if their key typed arguments e are unaffected by the update, i.e. $e^{new} = e^{old} = e$, they can also avoid recomputation as their delta expressions simplify to $\text{sng}(e, \delta_X(r))$ and $\text{fromK}(e, \delta_X(r))$, respectively. It is this observation that motivates the use of shredding for enabling incrementalization, as it replaces key-nested collections with labels which remain static throughout the execution while only their definitions are subject to updates.

Example 16. *In the following we present the result of delta derivation for the two kinds of*

$$\delta_X(X) = \Delta X \qquad \delta_X(Y) = 0 \qquad \delta_X(c) = 0$$

$$\delta_X(\langle r_1, r_2\rangle) = \langle \delta_X(r_1), \delta_X(r_2)\rangle \qquad \delta_X(r.\_i) = \delta_X(r).\_i$$

$$\begin{aligned}\delta_X(\text{let } Y := r_1 \text{ in } r_2) = \text{let } Y := r_1 \text{ in } \quad & \text{let } \Delta Y := \delta_X(r_1) \text{ in} \\ & (\delta_X(r_2) + \delta_Y(r_2) + \delta_X(\delta_Y(r_2)))\end{aligned}$$

$$\delta_X(r_1 + r_2) = \delta_X(r_1) + \delta_X(r_2) \qquad \delta_X(-r) = -\delta_X(r)$$

$$\delta_X(r_1 * r_2) = r_1 * \delta_X(r_2) + \delta_X(r_1) * r_2 + \delta_X(r_1) * \delta_X(r_2)$$

$$\delta_X(r_1 \cdot r_2) = r_1 \cdot \delta_X(r_2) + \delta_X(r_1) \cdot r_2 + \delta_X(r_1) \cdot \delta_X(r_2)$$

$$\delta_X(\text{sum}(r)) = \text{sum}(\delta_X(r)) \qquad \delta_X(\{x \Rightarrow r\}) = \{x \Rightarrow \delta_X(r)\}$$

$$\delta_X(\neg(r)) = \neg(r + \delta_X(r)) - \neg(r) \qquad \delta_X(p(x)) = 0$$

$$\delta_X(\text{sng}(e, r)) = \text{sng}(e^{new}, \delta_X(r)) + \text{sng}(e^{new}, r) - \text{sng}(e^{old}, r)$$

$$\delta_X(\text{fromK}(e, r)) = \text{fromK}(e^{new}, \delta_X(r)) + \text{fromK}(e^{new}, r) - \text{fromK}(e^{old}, r)$$

Figure 5.1 – Delta derivation rules for the constructs of our core calculus

nesting introduced in the previous section.

$$\begin{aligned}
\delta_X(\text{nestR}) &= \text{for}(\langle k_1, k_2\rangle \leftarrow \Delta X \textit{ if } p(k_2)) \text{ yield } k_1 \mapsto \text{sng}(k_2) \\
\delta_X(\text{nestK}) &= \text{for}(\langle k_1, k_2\rangle \leftarrow \Delta X) \text{ yield } \langle k_1, \text{toK}(\text{group}^{new}(k_1))\rangle \\
&\quad + \text{for}(\langle k_1, k_2\rangle \leftarrow X) \\
&\qquad \text{collect sng}(\langle k_1, \text{toK}(\text{group}^{new}(k_1))\rangle) \\
&\qquad\quad - \text{sng}(\langle k_1, \text{toK}(\text{group}(k_1))\rangle),
\end{aligned}$$

where group$^{new}(k_1)$ *operates on* $X^{new} = X + \Delta X$*, as opposed to* X*. While* nestR *(ring nesting) admits a delta query that only has to process the input update* ΔX*, in the case of* nestK *(key nesting) a change in* X *triggers the full replacement of nested collections whose content gets modified. We will show in the next section that upon shredding, one can avoid reevaluating inner collections when performing delta processing for key nesting as well.*

5.3 The Shredding Transformation

Shredding replaces by labels the ring types appearing within the key type of a mapping $\mathbb{K} \to \mathcal{R}$, and it associates a shredding context consisting of the corresponding label dictionaries. In particular, for every occurrence of a **RKey**$(\mathcal{R}')$ type we will have a dictionary **Label** $\to \mathcal{R}'$ in its shredding context, and this is done recursively for all the **RKey**$(\cdot)$ types inside $\mathcal{R}'$ as well. We define below the shredding contexts of types:

$$(\mathcal{R}_1 \times \mathcal{R}_2)^\Gamma = \mathcal{R}_1^\Gamma \times \mathcal{R}_2^\Gamma \qquad (\mathbb{K} \to \mathcal{R})^\Gamma = \mathbb{K}^\Gamma \times \mathcal{R}^\Gamma$$
$$(\mathbb{K}_1 \times \mathbb{K}_2)^\Gamma = \mathbb{K}_1^\Gamma \times \mathbb{K}_2^\Gamma \qquad (\mathbf{RKey}(\mathcal{R}))^\Gamma = (\mathbf{Label} \to \mathcal{R}) \times \mathcal{R}^\Gamma$$

while primitive types get shredded to an empty context $\varnothing : \texttt{Unit}$. To improve the presentation we often omit the empty contexts.

We introduce a transformation over expressions $r : \mathcal{R}$ in our calculus, that generates an expression $r^F : \mathcal{R}$ producing the result in shredded form, as well as an additional expression computing its shredding context $r^\Gamma : \mathcal{R}^\Gamma$. Due to the way we designed our language, most of its constructs are unchanged by the $(\cdot)^F$ transform, with the only exceptions being $\text{toK}(r)$ and $\text{fromK}(e)$.

When shredding is performed, instead of boxing ring values, $\text{toK}(r)$ produces a label ℓ based on the values of the free variables of expression r, while the companion shredding context $(\text{toK}(r))^\Gamma$ consists of a label definition $\ell \mapsto r^F : \mathbf{Label} \to \mathcal{R}$, along with the shredding context of r itself r^Γ.

For example, upon shredding query nestK in section 5.2.1, the subexpression $\text{toK}(\text{group}(k_1))$ constructs a label ℓ based on the value of k_1, as opposed to evaluating $\text{group}(k_1)$ and boxing the resulting collection. Moreover, as we will discuss in section 5.3.1, its shredding context will contain the label definition $\{\ell \Rightarrow \text{group}(\ell.k_1)\}$, where $\ell.k_1$ denotes the extraction of the free variable k_1 from the label. Since $\text{group}(k_1)$ no longer contains any key-nested ring values, we typically omit its corresponding empty shredding context.

Whenever $\text{toK}(\cdot)$ is applied to different expressions that have the same set of free variables we provide an additional static index ι, as in $\text{toK}_\iota(r)$, which gets factored into the constructed label along with r's free variables. This way we avoid the possibility of assigning the same label to different shredded ring values.

On the other hand, the $(\cdot)^F$ transform turns the construct for unboxing ring values $\text{fromK}(e)$, where $e : \mathbf{RKey}(\mathcal{R})$, into $\text{fromK}(e^F, e^{\Gamma_1})$. It essentially provides it with the corresponding dictionary from the shredding context e^Γ to be used for resolving the labels found in e^F, where e^{Γ_1} denotes the first component of e^Γ.

Example 17. *Let us consider the flattening query* $\text{flt}(\cdot)$ *in Example 13 which takes as input nested collection* $X : Bag(\boldsymbol{RKey}(Bag(\mathbb{K})))$*. Upon shredding the inner collections in* X *become labels, and their definitions are stored in dictionary* $X^\Gamma : \boldsymbol{Label} \to Bag(\mathbb{K})$*, whereas the* fromK *occurrence in* $\text{flt}(\cdot)$ *expression gets adjusted correspondingly to also reference* X^Γ*:*

$$\text{flt}(X) = \text{for}(x \leftarrow X) \text{ collect } \text{fromK}(x, X^\Gamma).$$

While initially x *would bind to the boxed inner collections in* X *and* fromK *would perform their unpacking, in the shredded variant* x *binds to the labels in the input, whereas* fromK *looks up* *their bag definition in X^{Γ}.*

The main benefit of shredding wrt. incrementalization comes in the form of removing the dependency on the input from the first argument of the sng and fromK constructs, thus opening their delta rules to the simplifications that render them free from having to perform re-evaluation. This comes as a result of turning those input-dependent key-nested ring expressions into labels, and separately managing their definition and corresponding input dependence as part of dictionaries within the shredding context.

5.3.1 Shredding Context

While the shredding transformation $(\cdot)^F$ we propose has minimal impact on the original expressions $r : \mathcal{R}$, i.e. it only requires changes wrt. the toK()/fromK() constructs, it does rely on an additional shredding context $r^{\Gamma} : \mathcal{R}^{\Gamma}$. Nonetheless, we remark that many of the constructs of our calculus operate only on the top level of collections, and in those cases the shredding context of their inputs is either preserved (eg. $-r$) or slightly restructured. The transformation rules for deriving shredding contexts are presented in Figure 5.2.

The shredding contexts we build closely follow the semantics of the constructs to which they correspond. In the case of tupling or projection, we also tuple or project the shredding contexts of their inputs.

Considering that the shredding context of mappings has two components, one for the keys and another for the ring values, the shredding context for constructing mappings (either singletons $\text{sng}(e,r)$ or infinite $\{x \Rightarrow r\}$) is likewise assembled out of those of its two inputs, where x^{Γ} is a shredding context variable to be bound to the context of the value bound by x. Moreover, when summing over the keys of a mapping via $\text{sum}(r)$, the component corresponding to its keys is also removed from the resulting shredding context, as those keys are no longer part of the output anymore.

Finally, the shredding context of key nested expressions $e : \textbf{RKey}(\mathcal{R})$ also has two components: a dictionary mapping the labels in their shredding to their definition, along with the context resulting from shredding the definition itself. This is reflected in the shredding context of $\text{toK}(r)$ which pairs the label dictionary $\{\ell \Rightarrow r^F\}$ with r^{Γ}. Analogously, when unpacking a key nested value using $\text{fromK}(e)$ we discard the dictionary e^{Γ_1} in its context since it has already been used earlier during its shredding transform ($(\text{fromK}(e))^F = \text{fromK}(e^F, e^{\Gamma_1})$) to resolve the labels in e^F.

Given mapppings $r_1 : \mathbb{K}_1 \to \mathcal{R}_1, r_2 : \mathbb{K}_2 \to \mathcal{R}_2$, we recursively define the shredding context of their product $(r_1 \cdot r_2) : (\mathbb{K}_1 \times \mathbb{K}_2) \to (\mathcal{R}_1 \cdot \mathcal{R}_2)$ based on the recursive definition of the product operator itself as:

$$r_1^{\Gamma} \odot r_2^{\Gamma} = \langle \langle r_1{}^{\Gamma_1}, r_2{}^{\Gamma_1} \rangle, r_1{}^{\Gamma_2} \odot r_2{}^{\Gamma_2} \rangle,$$

$$
\begin{array}{llll}
(\langle r_1, r_2 \rangle)^\Gamma = \langle r_1^\Gamma, r_2^\Gamma \rangle & (r.\_i)^\Gamma = r^\Gamma.\_i & (-r)^\Gamma = r^\Gamma & (p(x))^\Gamma = \varnothing \\
(\text{let } Y := r_1 \text{ in } r_2)^\Gamma = & \text{let } Y^F := e_1 \quad \text{in} & \text{let } Y^\Gamma := e_2^\Gamma \quad \text{in} & e_2^\Gamma \\
(\text{sng}(e, r))^\Gamma = \langle e^\Gamma, r^\Gamma \rangle & (\text{sum}(r))^\Gamma = r^{\Gamma_2} & \{x \Rightarrow r\}^\Gamma = \langle x^\Gamma, r^\Gamma \rangle & \\
(r_1 + r_2)^\Gamma = r_1^\Gamma \cup r_2^\Gamma & (r_1 \cdot r_2)^\Gamma = r_1^\Gamma \odot r_2^\Gamma & (r_1 * r_2)^\Gamma = \langle r_i{}^{\Gamma_1}, r_1{}^{\Gamma_2} \odot r_2{}^{\Gamma_2} \rangle & \\
 & (\text{fromK}(e))^\Gamma = e^{\Gamma_2} & (\text{toK}(r))^\Gamma = \langle \{\ell \Rightarrow r^F\}, r^\Gamma \rangle &
\end{array}
$$

Figure 5.2 – Transformation rules for deriving shredding contexts

where the empty shredding context acts as a neutral element. In particular, the first context component, which corresponds to keys $\langle k_1, k_2 \rangle : (\mathbb{K}_1 \times \mathbb{K}_2)$ in the resulting mapping, is obtained by pairing the key components of the input shreddings $r_1{}^{\Gamma_1}, r_2{}^{\Gamma_1}$.

The closely related multiplication operator $r_1 * r_2$ relies on the same procedure for deriving the second component of its shredding context ($r_1{}^{\Gamma_2} \odot r_2{}^{\Gamma_2}$). With respect to the first component, in order to preserve correctness it requires that the shredding contexts of its operands either have the same first components (i.e. $r_1{}^{\Gamma_1} = r_2{}^{\Gamma_1}$) or one of them is a variable (as x^Γ from the shredding of $\{x \Rightarrow r\}$). In the latter case the variable x^Γ gets bound to the concrete value provided by the other operand's shredding context.

For deriving the shredding context of the addition operator we introduce the union operation $r_1^\Gamma \cup r_2^\Gamma$ over shredding contexts, which performs union between corresponding dictionaries in r_1^Γ, r_2^Γ (as they have the same type). The goal of this operation is twofold: (i) to collect the label definitions for all the labels that may appear in the result of $r_1 + r_2$, and (ii) to enforce that a label appearing in both r_1, r_2 has the same definition in both r_1^Γ, r_2^Γ (and flag the opposite case as an error). While the shredding contexts obtained as a result of our derivation are guaranteed not to introduce any inconsistent label definitions, they may still occur as part of arbitrary input contexts.

Example 18. *Shredding bag $X : Bag(\mathbb{K}_1 \times \mathbb{K}_2)$ associates a shredding context $X^\Gamma = \langle d_1, d_2 \rangle : \mathbb{K}_1^\Gamma \times \mathbb{K}_2^\Gamma$, where we omitted X^Γ's empty value component corresponding to integer multiplicities. By expanding the definition of the* nestK *query in Section 5.2.1 and applying the rules for deriving shredding contexts we obtain:* $\text{nestK}^\Gamma = \langle d_1, \langle \{\ell \Rightarrow \text{group}(\ell.k_1)\}, d_2 \rangle \rangle$*, where we used the fact that* $(\text{group}(\ell.k_1))^\Gamma = d_2$.

Correctness. From shredded results one can still recover the original nested value by recursively replacing each label with a boxed ring value containing the label's definition from the corresponding shredding context. For instance, the nested result in our example can be recovered via:

$$\begin{aligned}\text{nestK} = \text{for}(\langle k_1, k2\rangle \leftarrow \text{nestK}^F)\\ \text{collect}\ \langle k_1, \text{toK}(\text{fromK}(k_2, \text{nestK}^{\Gamma_2}))\rangle,\end{aligned}$$

where nestK^F represents the flat output of the shredded version of the query.

If we denote by $\mu_{\mathcal{R}}(X^F, X^\Gamma)$ the type-indexed operator that applies this label substitution for values $X : \mathcal{R}$, one can easily show via structural induction on the constructs of our calculus that:

Theorem 14. *Given any expression $e : \mathcal{R}$, nesting back the result of its shredded version produces the same value as the original expression, i.e.: $e = \mu_{\mathcal{R}}(e^F, e^\Gamma)$.*

5.4 System architecture

The architecture of SLeNDer is split into front-end and back-end components (see Figure 5.3), both implemented in Scala. It takes as input nested queries and produces either Spark batch-processing code for their equivalent shredded representation, or Spark trigger programs for incrementalizing their results wrt. mini-batch updates for their inputs. In addition, it provides an interpreted evaluation mode as well as a runtime library for single-node execution.

The frontend operates on a typed deep embedding of the nested ring calculus presented in Section 5.2. In order to provide more convenience when defining queries, this language is extended with several constructs, like for-comprehensions, which are nonetheless de-sugared to the core constructs of the calculus. The embedding uses native Scala types for modeling primitive types and tuples, i.e. key types, while for ring types we define appropriate ring structures by leveraging Scala's implicit classes functionality. As a result, we can easily extend the type system as needed with additional types that exhibit a ring structure, and we can also reuse the host compiler for type-checking the given queries.

The frontend is designed around two main tasks: i) simplifying queries by applying constant folding or the ring simplification rules (i.e. neutral element, absorbing element, etc.) and ii) deriving shredded programs based on the shredding transformation described in the previous section.

The backend performs recursive incrementalization and materialization over normalized queries. These transformations are enabled by a specialized intermediate representation which we refer to as Recursive A-normal form (detailed in Section 5.4.1), as well as an untyped lower-level DSL, whose type annotations are maintained as data in the IR nodes. This facilitates the definition and application of optimization rules without the obligation of providing proofs of type-preservation, which is a non-trivial task for many common optimization rules. Moreover, it streamlines syntax trees as well as their optimization by working with n-ary operators (as opposed to just binary) for product or addition constructs, considering that they are

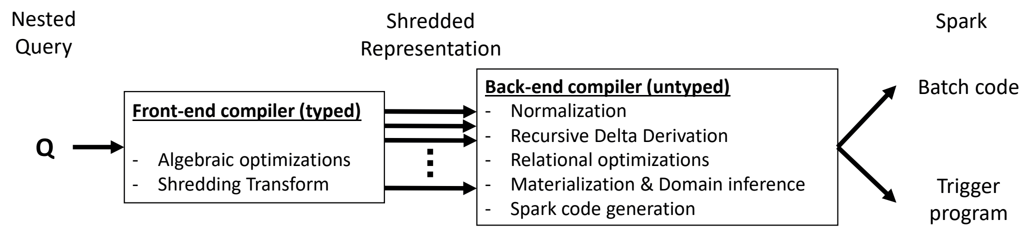

Nested Query
Q
Front-end compiler (typed)
- Algebraic optimizations
- Shredding Transform
Shredded Representation
Back-end compiler (untyped)
- Normalization
- Recursive Delta Derivation
- Relational optimizations
- Materialization & Domain inference
- Spark code generation
Spark
Batch code
Trigger program

Figure 5.3 – The architecture of SLeNDer.

both associative and commutative. Finally, it flattens the structure of tuples by associating names to fields, in contrast to addressing tuple members via their position, which is the case in the frontend. These design choices greatly simplify transformations that reorder products of operands, as needed when separating the parts of a query that depend on the delta from those that only depend on the input relations.

As a final step, it applies standard relational optimizations (e.g. pushing aggregates, selections) before generating Spark code, which leverages common collection primitives like `.filter(·)` or `.reduceByKey(·)`. At this point we do not apply classical compiler optimizations like common subexpression elimination or inlining, as we expect their impact to be limited considering the high level nature of the code we generate.

5.4.1 Recursive ANF

The back-end compiler of SLeNDer uses an intermediate representation which we refer to as Recursive A-normal form (ANF). It is designed to enable the modular specification and application of classic relational optimization rules, as well as of recursive incrementalization and materialization. While regular ANF breaks down a given expression into basic subexpressions consisting of a functional (operator) application over constants or variables, Recursive ANF leverages the algebraic properties of operators to lazily generate a tree of subexpressions exploring alternative execution plans of interest for the given expression. This aspect of code generation is essential for domain specific languages (like relational calculus or linear algebra) whose operators are part of rich algebraic structures, and where the choice between equivalent re-writings of an expression has substantial consequences on its execution time.

Example 19. *Let us consider the following query performing a three-way join over binary relations $R, S,$ and T:*

$$\begin{aligned} Q = &\text{for}(\langle x, y\rangle \leftarrow R) \text{ collect}\\ &\quad \text{for}(\langle z, w\rangle \leftarrow S) \text{ collect}\\ &\quad \text{for}(\langle v, t\rangle \leftarrow T \textit{ if } x > 5 \;\&\&\; y == z \;\&\&\; w == v \;\&\&\; t < 12) \text{ yield } \langle x, t\rangle \end{aligned}$$

One possible ANF representation of this query is then:

$$\begin{aligned}
Q = \text{let } Q_1 &:= \text{for}(\langle z,w\rangle \leftarrow S) \text{ collect for}(\langle v,t\rangle \leftarrow T) \text{ yield } \langle z,w,v,t\rangle \text{ in}\\
\text{let } Q_2 &:= \text{for}(\langle x,y\rangle \leftarrow R) \text{ collect for}(\langle z,w,v,t\rangle \leftarrow Q_1) \text{ yield } \langle x,y,z,w,v,t\rangle \text{ in}\\
&\text{for}(\langle x,y,z,w,v,t\rangle \leftarrow Q_2 \textit{ if } x > 5 \;\&\&\; y == z \;\&\&\; w == v \;\&\&\; t < 12) \text{ yield } \langle x,t\rangle
\end{aligned}$$

We classify the equivalence rules that we use in generating the expressions tree of Recursive ANF into *unidirectional* vs *bidirectional* rules. The former include rules like pushing of filtering or aggregates, which are deemed beneficial irrespective of the workload, while the latter include rules like join or matrix multiplication re-ordering where the desired version depends on workload characteristics. While the unidirectional rules are always applied, it is the bidirectional ones which are responsible for the branching structure of the Recursive ANF representation of an expression.

Example 20. *Upon applying the unidirectional rules on the ANF representation of our three-way join example above we get:*

$$\begin{aligned}
Q = \text{let } Q_T &:= \text{for}(\langle v,t\rangle \leftarrow T \textit{ if } t < 12) \text{ yield } \langle v,t\rangle \text{ in}\\
\text{let } Q_{ST} &:= \text{for}(\langle z,w\rangle \leftarrow S) \text{ collect for}(\langle v,t\rangle \leftarrow Q_T \textit{ if } w == v) \text{ yield } \langle z,t\rangle \text{ in}\\
\text{let } Q_R &:= \text{for}(\langle x,y\rangle \leftarrow R \textit{ if } x > 5) \text{ yield } \langle x,y\rangle \text{ in}\\
&\text{for}(\langle x,y\rangle \leftarrow Q_R) \text{ collect for}(\langle z,t\rangle \leftarrow Q_{ST} \textit{ if } y == z) \text{ yield } \langle x,t\rangle,
\end{aligned}$$

as a result of pushing selections to the base relations and projecting away the unnecessary fields from the intermediate results.

After also applying the bidirectional rules we end up with the following Recursive ANF representation:

$$\begin{aligned}
Q = \text{let } Q_T &:= \text{for}(\langle v,t\rangle \leftarrow T \textit{ if } t < 12) \text{ yield } \langle v,t\rangle \text{ in}\\
\text{let } Q_{ST} &:= \text{for}(\langle z,w\rangle \leftarrow S) \text{ collect for}(\langle v,t\rangle \leftarrow Q_T \textit{ if } w == v) \text{ yield } \langle z,t\rangle\\
&:= \text{for}(\langle v,t\rangle \leftarrow Q_T) \text{ collect for}(\langle z,w\rangle \leftarrow S \textit{ if } w == v) \text{ yield } \langle z,t\rangle \text{ in}\\
\text{let } Q_R &:= \text{for}(\langle x,y\rangle \leftarrow R \textit{ if } x > 5) \text{ yield } \langle x,y\rangle \text{ in}\\
&\text{for}(\langle x,y\rangle \leftarrow Q_R) \text{ collect for}(\langle z,t\rangle \leftarrow Q_{ST} \textit{ if } y == z) \text{ yield } \langle x,t\rangle\\
= \text{let } Q_{RT} &:= \text{for}(\langle x,y\rangle \leftarrow Q_R) \text{ collect for}(\langle v,t\rangle \leftarrow Q_T) \text{ yield } \langle x,y,v,t\rangle\\
&:= \text{for}(\langle v,t\rangle \leftarrow Q_T) \text{ collect for}(\langle x,y\rangle \leftarrow Q_R) \text{ yield } \langle x,y,v,t\rangle \text{ in}\\
&\text{for}(\langle z,w\rangle \leftarrow S) \text{ collect for}(\langle x,y,v,t\rangle \leftarrow Q_{RT} \textit{ if } y == z \;\&\&\; w == v) \text{ yield } \langle x,t\rangle\\
= \text{let } Q_{RS} &:= \text{for}(\langle x,y\rangle \leftarrow Q_R) \text{ collect for}(\langle z,w\rangle \leftarrow S \textit{ if } y == z) \text{ yield } \langle x,w\rangle\\
&:= \text{for}(\langle z,w\rangle \leftarrow S) \text{ collect for}(\langle x,y\rangle \leftarrow Q_R \textit{ if } y == z) \text{ yield } \langle x,w\rangle \text{ in}\\
&\text{for}(\langle v,t\rangle \leftarrow Q_T) \text{ collect for}(\langle x,w\rangle \leftarrow Q_{RS} \textit{ if } w == v) \text{ yield } \langle x,t\rangle.
\end{aligned}$$

We remark, that even though certain sub-expressions appear multiple times within the Recursive ANF representation (eg. Q_R, Q_T), they only get expanded once. In fact the Recursive ANF can be seen as a DAG of possible evaluation plans for a given expression, built from two kind of nodes: (i) operator nodes corresponding to the typical ANF nodes applying an operator over one or two intermediate results, and (ii) alternation nodes enumerating the equivalent ways of re-writing the current subexpression (in terms of the corresponding operator nodes).

Finally, we note that the explosion in the size of the Recursive ANF can be controlled via expansion strategies that can choose to explore only left- (right-) deep evaluation plans, or that simply ignore certain alternatives, such as those producing equivalent Cartesian products.

In SLeNDer we use Recursive ANF both during the query code generation phase as well as when generating trigger code for incrementalizing views, more precisely when deciding which subexpressions of a delta query to materialize and thus reuse across delta applications.

The expansion into a set of equivalent expressions that the Recursive ANF representation captures is not unlike the exploration that a cost-based query optimizer performs in its search for an optimal query plan. In our work we recognize the fact that the search space inspected in the process has applications beyond just cost-based optimizations, and thus Recursive ANF formalizes it and makes it available in a manner that is independent from its ultimate use. For example, our particular instance of the materialization problem fits within the larger area of partial evaluation of programs in the case when different inputs become available at different times, and thus the Recursive ANF representation has wider applicability in DSL compilers beyond the concerns of SLeNDer. Moreover, considering that the search space is the result of often complex re-write rules, from a development point of view it is desirable to describe these rules and more importantly make sure that their application is sound only once, and then make the resulting exploration available to the different compiler passes that might need it.

5.5 Experiments

We evaluate the performance of shredded queries compared to the original ones both in an offline and online (incremental) setting. In offline scenarios we show that shredding is effective in removing skew from inner collections and thus load balance their processing across all the nodes available, whereas for online scenarios we show that keeping the result of nested queries fresh by incrementally maintaining their shredded counterparts via Recursive IVM is at least an order of magnitude faster than re-evaluation.

Experimental Setup We run our experiments on a cluster consisting of 90 servers, each with 2 Intel Xeon E5-2630L @ 2.40GHz CPUs (each CPU has 6 cores, and each core has 2 hardware threads), 15MB of cache, 256GB of DDR3 RAM, connected via a full-duplex 10GbE network and running Ubuntu 14.04.2 LTS, Spark 1.5.0 and YARN 2.7.1. We generate Scala programs for running on Spark and compile them using Scala 2.10.6.

Flat-to-nested	Q1	Build the hierarchical relation Customer - Order - Lineitem, where for each order we record the date, and for each lineitem we record the part name and the quantity.	3
	Q2	Compute the list of Customers of each Supplier according to their orders.	3
	Q3	For every part compute the list of suppliers and the list of customers.	4
Nested-to-flat	Q4	Given the hierarchical relation Customer - Order - Lineitem (Q1), compute for each customer and each part she bought the total quantity per year.	0
	Q5	Given the result of Q3, compute for every part the number of customers without a national supplier.	0
Nested-to-nested	Q6	Given the list of Customers of each Supplier (Q2), compute a list of suppliers per customer.	1
	Q7	Given the result of Q3, compute for every country the list of exported parts.	1

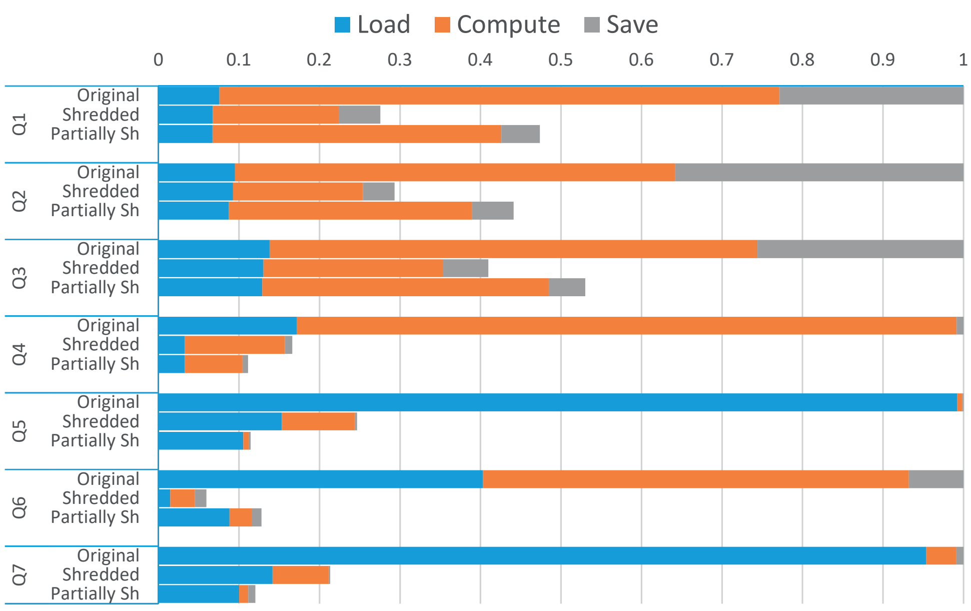
Table 5.1 – Description of the queries included in our workload, along with the number of joins they require.

Queries: The queries (Q1-Q7) in our workload are described at a high level in Table 5.1 (for full definition see appendix A.5). They were designed based on common tasks involving nested data like establishing hierarchical relations (Q1), computing aggregates across nesting levels (Q4) or inverting the index of grouped data (Q6), and although they make use of the TPC-H schema they have no relation to the queries in the TPC-H benchmark.

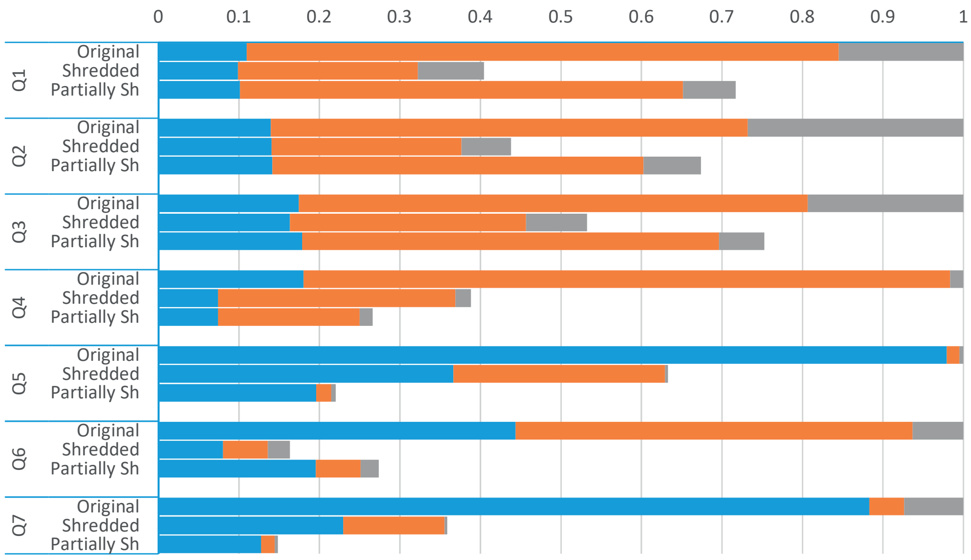
We include three kinds of queries depending on the type of data they consume/produce: flat-to-nested (Q1-Q3), nested-to-flat (Q4-Q5), and nested-to-nested (Q6-Q7). We run the flat-to-nested queries over TPC-H data (with scaling factor of 500), while the rest use their results as nested input. For generating skewed data we use the TPC-H generator proposed by [16] (with a *Zipf* skew parameter of 2), which we modified in order to fix the proportion of values in a specific column that are drawn from the skewed distribution, while the remaining values are taken from a uniform distribution.

5.5.1 Deep scaling

We measure the ratio between processing times with and without shredding for the queries in our workload and in Figures 5.4 and 5.5 we breakdown the cost in terms of the time required to load the data (LOAD), evaluate the query (COMPUTE) and finally save the result (SAVE). We vary the percentage of skewed data between 20% and 0% (no skew) in order to capture the

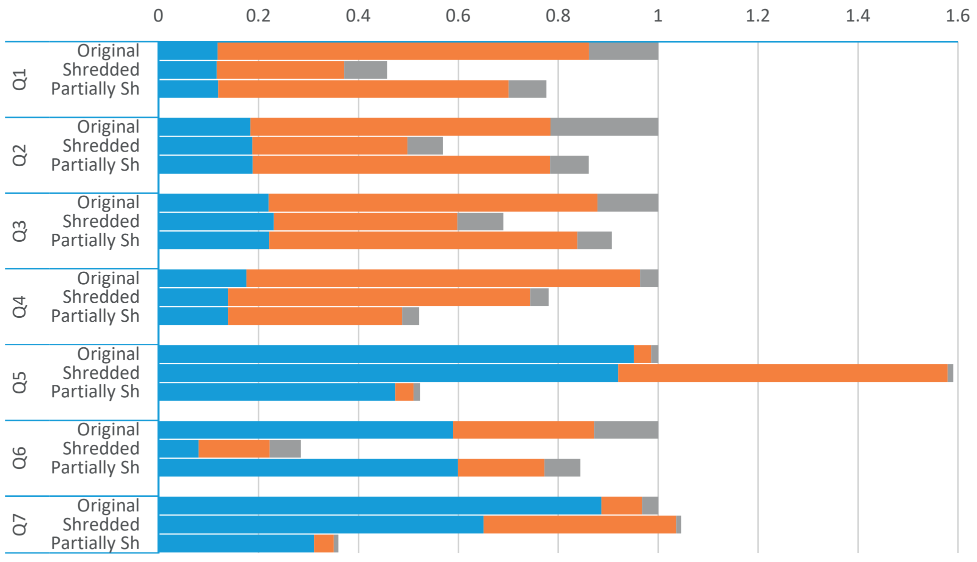


(a) 20%

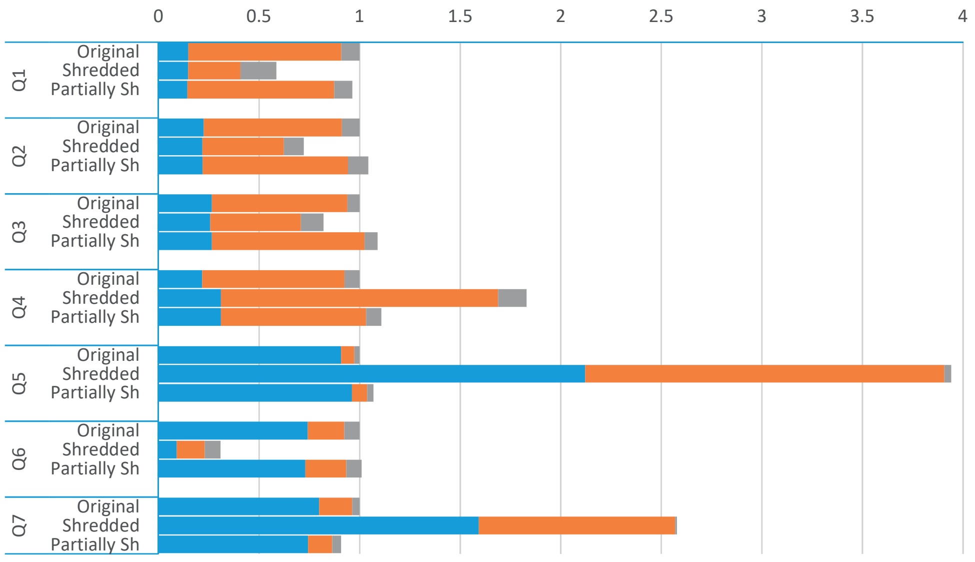


(b) 10%

Figure 5.4 – Running times ratio of Shredded and Partially Shredded queries vs the original queries for high percentages of skewed data (20% and 10%).



(a) 5%



(b) 0%

Figure 5.5 – Running times ratio of Shredded and Partially Shredded queries vs the original queries for low percentages of skewed data (5% and 0%).

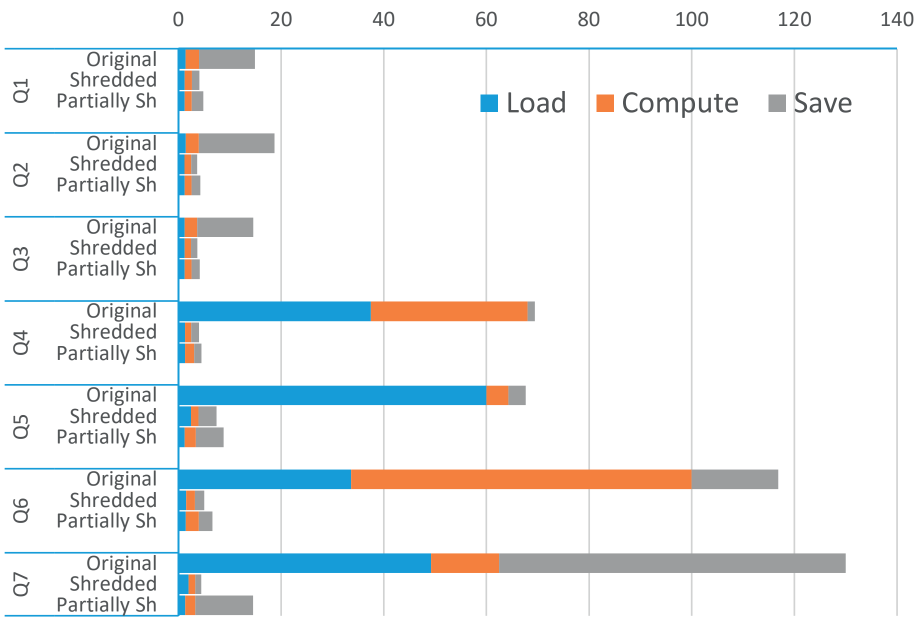


Figure 5.6 – Ratio between the running time of the longest and the median tasks aggregated per stage (when 20% of the data is skewed).

performance of the queries under different levels of load imbalance, as well as to isolate the overheads of shredding when skew is relatively small or absent. Moreover, in order to illustrate the effectiveness of shredding in reducing skew we present in Figure 5.6 the ratio between the running times of the longest and the median tasks within Spark stages, aggregated over the three phases of the query evaluation we consider.

We divide our discussion based on the type of the query. For **flat-to-nested** queries (Q1-Q3), we see a benefit across the board from full-shredding, both in evaluation and output times (as expected, loading times are unaffected). The advantage is maintained also when no skew is present (Figure 5.5b), since producing results in shredded form requires at least one less join compared to the nested version.

The shredded variants of **nested-to-flat** queries (Q4-Q5) also enjoy considerably smaller loading and processing times in the presence of high skew (Figure 5.4). Since they are able to push aggregations down to the shredded collections, the processing of large nested collection ends up distributed across multiple nodes, all the while significantly reducing the amount of data that needs to be joined back together in order to produce the final result. By contrast, the original queries must first pay the high price of loading the large inner collections, and then have to process each on a single node.

However, in the absence of skew, shredded nested-to-flat queries are at a disadvantage (Figure 5.5b), since they have to read more data, i.e. for each nested collection in the original input

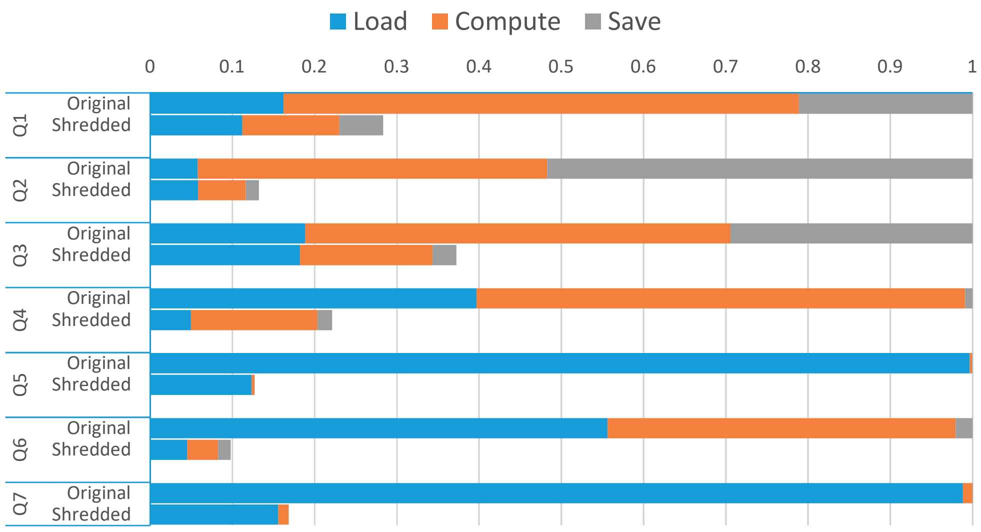


Figure 5.7 – Running times ratio of shredded vs original queries for low cardinality top-level collections (10 top-level records).

they have to read several label definitions containing just a few tuples as they are partitioned across the cluster, each with its own space overhead. Moreover, they also need to perform an extra join as they match the shredded tuples of their inputs with the intermediate results obtained by processing the corresponding shredding context.

Finally, shredded **nested-to-nested** queries (Q6-Q7) benefit from both the ability to balance the workload by pushing processing down to shredded collections as well as avoid the final join or grouping of the result, since their output is kept in shredded form as well. In fact the shredded version of Q6 manages to avoid any kind of reshuffling in its evaluation and as a result outperforms the original query even in the absence of skew (Figure 5.5b).

As exemplified by Q3 and Q5, there is a tradeoff between the savings obtained when producing shredded results due to the avoidance of an extra join, and the additional costs incurred when consuming these results, and finally having to perform the join between top-level records and intermediate results corresponding to the processing of label definitions. However, **partial shredding** addresses this issue as it creates distributed label definitions only for those inner collections that are large enough to provoke load imbalance. This way we can shift back the additional join to the query producing the nested output, while the query consuming it has to match only the relatively few labels that have been introduced by shredding. More importantly, all this is done without the risk of generating load imbalance. As a consequence, the *partially* shredded versions of flat-to-nested queries (Q1-Q3) have higher evaluation costs than the fully shredded variants, while for the rest of the queries consuming their results we see substantial improvements.

Considering that none of the shredding strategies preform optimally for all queries and for all levels of skew we argue that choosing the best shredding strategy should be left to the query optimizer which can take the appropriate decision based on the structure of the query in terms of its nesting and join pattern, along with statistics regarding the distribution of values within grouping columns.

Finally, we also show that shredding can dramatically increase performance when processing **top level collections with low cardinality** but large inner collections even in the absence of skew (Figure 5.7), with speedups ranging between 2.68x and 10.19x.

5.5.2 Incremental Evaluation

We compare the speedups provided by (recursive) incremental view maintenance wrt. full re-evaluation (see Figure 5.8), in terms of the time it takes to process the entire input data divided in 50 equal batches, with each batch randomly partitioned between the available workers and preloaded in memory before the start of the experiment. The streams are obtained by interleaving batches of insertions to the base relations in a round-robin fashion. We run one Spark job per batch to refresh the results, while the auxiliary materialized views are updated lazily when used. We rely on simple heuristics for deciding how to partition auxiliary views and use the key of the join that references the view. In the case of views referenced by multiple joins we use the key with the highest cardinality.

If the input is nested, the updates will also consist of nested tuples, while the output is incrementally maintained in shredded form, with the final nested result being computed only on demand. We argue that this final step should be performed lazily only for the values that end up being outputted, while the memory/storage layout should remain shredded, as it facilitates efficient further updating and processing.

Unfortunately, neither Streaming Spark [73], nor the incremental evaluation engine of Spark (Spark Structured Streaming) support joins between two streaming datasets with full semantics, and while DBToaster [36, 54] implements a recursive incrementalization strategy that does support joins, it currently cannot handle queries that process nested data. Therefore, we compare our approach against Spark Streaming only for queries that do not join dynamic datasets (Q4, Q5 and Q7), whereas for the rest we compare against a standard incrementalization (IVM) technique that materializes the intermediate results of joins and uses them in propagating delta changes up the query plan.

The speedups obtained through recursive incrementalization by the trigger programs we generate range from 11.05x to 21.93x, when compared to reevaluation. By contrast, standard IVM manages speedups of only 1.49x to 16.89x. These speedups are supported by the fact that the incrementalization techniques maintain query results while processing overall much

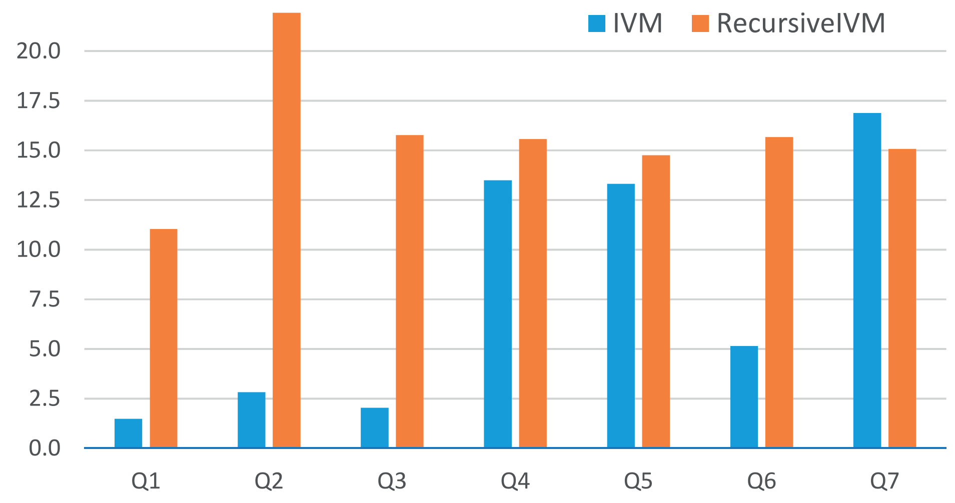


Figure 5.8 – Speedups of incremental evaluation of queries vs recomputation.

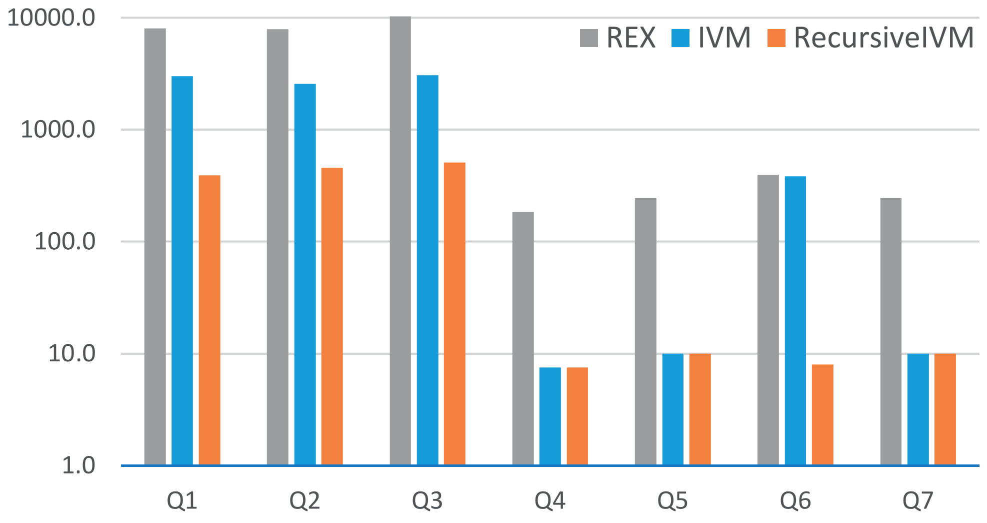


Figure 5.9 – Number of tuples (x 10 million) processed during the maintenance of queries via re-execution (REX), vs. incremental maintenance (IVM), vs. Recursive IVM.

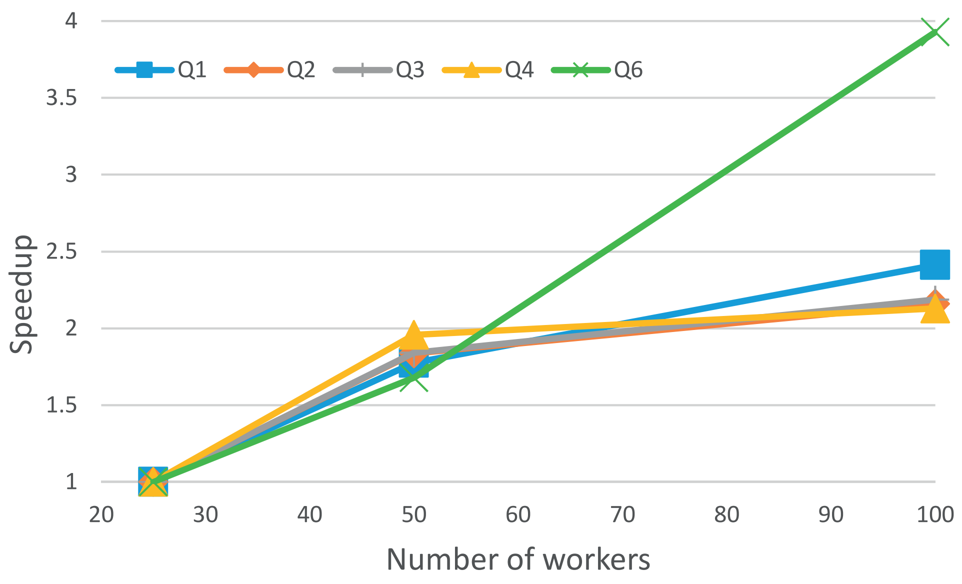


Figure 5.10 – Speedups of Recursive IVM for different number of workers (25, 50 and 100).

fewer tuples than by reevaluation, as highlighted in Figure 5.9. We remark that the queries where Recursive IVM significantly outperforms standard IVM (up to 7.7x faster) are exactly those that execute several joins, whereas for queries with no joins (Q4-Q5) or those that join against a static table (Q7), the two techniques behave similarly.

Finally, we show in Figure 5.10 the speedups obtained by Recursive IVM when processing batches on 25, 50 and 100 workers. Query Q6 achieves almost linear speedup considering that its trigger program requires no reshuffling for updating shredded results. The performance with 100 workers more than doubles when compared to the 50 workers data point, as the pressure on the memory resources of each node diminishes, leading to less time wasted on memory management. On the other hand, queries Q1-Q4 reach at most a 2.41x improvement for 100 workers vs. 25. This is a consequence of the additional intermediate views (i.e. materialized delta queries) that they have to maintain, and ultimately have to join against in order to refresh their output. We omitted from Figure 5.10 queries Q5 and Q7 since the amount of work required to process one of their batches was relatively small thus leading to poor scaling behavior.

6 Related Work

6.1 Incremental processing for nested queries

The **nested data model** has been thoroughly studied in the literature over multiple decades and has enjoyed a wide adoption in industry among NoSQL data management systems and 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 [25], or propagate those changes based on auxiliary data structures designed to track the lineage of tuples in the view [21, 24, 33, 53]. The use of dedicated data-structures as well as custom update languages make it extremely difficult to further apply query optimizations on top of these techniques.

Delta processing was originally proposed for datalog programs [30, 31] but it is even more natural for algebraic query languages [7, 9, 13, 15, 27, 38, 57, 60, 71], especially relational algebra on bags, 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 [35, 74]. Compared to [35] 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.

In our implementation we adopt an eager evaluation strategy, but a broad discussion of the implications of this choice on the view maintenance process can be found in the literature [18, 19, 31, 61, 75].

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* [2–5, 41], 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. Similarly, Maier et. al. [48] leveraged the existing infrastructure in a functional reactive programming framework to allow for incremental reactivity over lists. As this techniques make few assumptions about their target language, they are applicable to a variety of languages ranging from Standard ML to C. Nonetheless, their generality comes at the price of significant runtime overheads for building dependency graphs. Moreover, while static solutions derive deltas that can be further optimized via global transformations, such an opportunity is mostly missed by dynamic approaches (the potential for optimizations is even more consequential when dealing with languages with powerful algebraic laws). For instance, one can combine factorization and the commutativity of operators to implement a more powerful form of the traditional optimization of common subexpression elimination. By contrast, this would not be applicable to general-purpose Turing-complete languages, as such algebraic laws do not exist on arbitrary data values.

When compared to a monolithic incremental view maintenance algorithm, an algebraic approach by delta-query rewriting has the additional advantage of re-using the query processing and evaluation infrastructure of the database system. One can thus leverage decades of progress on query processing technology for incremental view maintenance and does not need to invent analogous technology – say, reinvent indexes for IVM.

Delta derivation has also been proposed in the context of incremental computation, initially only for first-order languages [56], and more recently it has been extended to higher-order languages [11]. Their work represents an initial step towards closing the gap between the applicability of static and dynamic approaches, and creating a unified field and understanding of incremental computation.

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. Moreover, when incrementalizing functional values $f : B^A$ they only allow functional updates $df : B^{A\times A}$, that satisfy an additional non-trivial constraint:

$$\begin{aligned} f(a) \oplus_B df(a, da) &= f(a \oplus_A da) \oplus_B df(a \oplus_A da, \varnothing) \\ &= f(a) \oplus_B \delta(f)(a, da) \oplus_B df(a \oplus_A da, \varnothing). \end{aligned}$$

In other words, if we consider $g(a) = df(a, \varnothing)$ to be an independent function, it follows that $df(a, da)$ must compute both the value of $g(a \oplus_A da)$ and $\delta(f)(a, da)$. This effectively leaves the problem of delta derivation to the user. This constraint along with their definition of $\oplus_{B^A}$, $(f \oplus_{B^A} df)(a) = f(a) \oplus_B df(a, \varnothing)$, guarantees that function application $\mathrm{app} : B^A \times A \to B$

can be easily incrementalized as follows:

$$\mathrm{app}(f \oplus_{B^A} df, a \oplus_A da) = \mathrm{app}(f, a) \oplus_B \mathrm{app}(df, \langle a, da \rangle).$$

By contrast, we allow any functional value $df : B^A$ to act as a change and in incrementalizing app we leverage the fact that functional values in a simply typed lambda language can be represented as closures. Thus, we can apply the δ derivation on their underlying term, just like we would on any other term of the language. In addition, we extend to simply-typed lambda calculi the concept of higher-order deltas (i.e deltas of deltas) which was shown to provide significant speedups when compared to traditional delta processing [36].

Memoization or function caching has also been used for incremental computation. Liu [46, 47] employs static dependency analysis in order to infer which subexpressions of a function do not depend on the input update. It then transforms the original program into an incremental version where those independent intermediate results are reused from the initial run. However, these transformations depend on a library of rewriting rules that apply only under very restrictive conditions.

Data streaming systems [1, 51] provide fast online analytics by employing techniques like approximate processing, load shedding, prioritization [67], on-the-fly aggregation [37], or specialized algorithms [20]. However these proposals have seen limited adoption as they do not preserve SQL semantics and are challenging to compose.

6.2 Shredding nested queries

While, 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 [17, 29, 34, 40, 63, 68], we are, to the best of our knowledge, the first to propose a generic and compositional **shredding** transformation that can handle any nested-to-nested queries[1], 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.

Previous approaches cover a range of semantics and serve various goals, from proving expressiveness/complexity results to off-loading query processing to backend database systems. However, many of them are global (or context dependent), i.e. in translating a construct they require a full (or partial) view of the query being translated, which makes them hard to extend with new language constructs or adapt to new applications, thus leading to a flurry of proposals, one for each new use-case.

The challenge of shredding nested queries has been initially addressed by Paredaens et al. [58], who propose a translation taking flat-to-flat nested relational algebra expressions into flat

[1] As opposed to only flat-to-flat or flat-to-nested queries.

relational algebra. Van den Bussche [68] 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 [17].

Shredding transformations have been studied more recently in the context of language integrated querying systems such as Links [44] and Ferry [28]. 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` [29], while Links relies on a normalization phase and handles only flat-to-nested expressions [17]. More importantly, none of the existing proposals translate NRC$^+$ queries to an efficiently incrementalizable language (for instance, the output language of Ferry includes the difference operator which cannot be maintained efficiently).

The challenges raised by working with potentially large nested collections have been primarily addressed at the storage level by the introduction of columnar storage formats, as the one proposed by Dremel [50]. While these formats store data in shredded form, and by doing so they enable large reductions in the costs of retrieving it (due to a compact layout and pushing projections), the processing of the data is still done in nested form, meaning that the relevant fields are still assembled back into (nested) records before being delivered to the querying engine. This means that they cannot avoid the imbalances inflicted by skewed inner collections. By contrast, the **shredding** transformation we developed produces shredded queries that operate directly on shredded data, thus enjoying better scalability.

In its efforts to extract as much parallel computation as possible out of programs manipulating nested data structures, Data Parallel Haskell [59] relies on a flattening transformation extending to a higher-order language the proposals of Blelloch et al. [8] and Suciu et al. [65]. However, they use arrays as their underlying data model, which lack many of the optimizations opportunities afforded by our collections with ring typed multiplicities, including those essential for incrementalization via delta derivation. Moreover, their scheme for addressing flattened data is based on random index lookups, and as such only appropriate for a shared memory multicore execution environment. By contrast, we target map-reduce frameworks where such access patterns would be prohibitively expensive.

7 Conclusions

Due to its familiar abstractions, collection programming has played a key role in lowering the entrance barrier for practitioners of many computer science fields when it comes to processing large datasets. In addition, a major factor in its wide adoption lies in its optimization potential, particularly in terms of parallelization. The fact that collection programming fits in a low parallel complexity class makes it especially suitable for scaling processing within massively distributed platforms.

In this work, we showed that collection programming is also amenable to *efficient* incrementalization via re-writing rules. In particular, we assessed the cost of collection programs based on a cost semantics proposal tailored to *nested* data, and we showed that the incrementalized versions of queries written in the positive fragment of the language are indeed cheaper than recomputation.

Moreover, we proved that the incrementalized versions of collection programs running over constant-size updates are in a lower complexity class than re-evaluation (NC_0 vs. TC_0). Considering that the computation model assumed by TC_0 allows for circuits with unbounded fan-in, while NC_0 precludes them, this class separation implies that incrementalization incurs less communication overhead, which is a major bottleneck in large-scale processing.

We also took an important step towards extending the reach of incrementalization techniques based on algebraic re-writings from classic first-order querying languages to higher-order (functional) languages. This is especially relevant for collection programming as it is often embedded within functional programming languages, but it also opens up the possibility of developing processing frameworks that can react to both dynamic datasets as well as dynamic query-sets (small changes to queries should not necessarily require the full re-evaluation of the workload as long as they can be captured as delta's of functional values). In particular, we show that given efficiently incrementalizable primitives, the simply-typed lambda calculus built around them admits a delta-transformation that produces efficient incremental versions of programs. To do so, we internalize the delta-transformation as a construct of the language and apply it as needed over the term component of closures.

In order to fulfill the promise of current collection processing systems of virtually unlimited scaling both in online and offline scenarios, special consideration is required towards the way inner collections are distributed among workers. The status-quo of sequentially processing each inner collection on a single node can result in severe slow-downs in the cases when their sizes are drastically different from one top-level record to the next. To that end we proposed the SLeNDer compilation framework which relies on *shredding* in order to translate a nested query into code that evenly load balances the computation of nested collections across cluster resources. In addition, shredding is also leveraged for generating trigger programs which efficiently incrementalize nested queries based on the technique of Recursive IVM.

Our experimental evaluation showed that shredding is indeed effective in eliminating skew wrt. the sizes of inner collections, and that applying updates via recursive incrementalization results in an order of magnitude speedups compared to re-evaluation. All this comes on top of already established benefits of shredding in terms of enabling optimizations across nesting levels as highlighted by previous work on query unnesting / decorrelation. Moreover, we show that *partial shredding* is effective in eliminating most of the overheads incurred when stitching back shredded results, while retaining its load balancing benefits wrt. large inner collections.

A Appendix

A.1 Incrementalizing IncNRC$^+$

A.1.1 The delta transformation

Proposition 3.2.1. *Given an* IncNRC$^+$ *expression* $h[R] : \mathbf{Bag}(B)$ *with input* $R : \mathbf{Bag}(A)$ *and update* $\Delta R : \mathbf{Bag}(A)$*, then:*

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

Proof. The proof follows by structural induction on h and from the semantics of IncNRC$^+$ constructs.

- For $h = R$, the result follows immediately.
- For $h \in \{\emptyset, p, \mathbf{sng}(x), \mathbf{sng}(\pi_i(x)), \mathbf{sng}(\langle\rangle), \mathbf{sng}^*(e)\}$ as the query does not depend on the input bag R we have $h[R \uplus \Delta R] = h[R]$ and the result follows immediately.
- For $h = \mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2$:

$$
\begin{aligned}
&[[(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2)[R \uplus \Delta R]]]_{\gamma;\varepsilon} = \\
&= \biguplus_{v \in [[e_1[R \uplus \Delta R]]]_{\gamma;\varepsilon}} [[e_2[R \uplus \Delta R]]]_{\gamma;\varepsilon[x:=v]} \\
&= \biguplus_{v \in [[e_1[R]]]_{\gamma;\varepsilon} \uplus [[\delta(e_1)[R,\Delta R]]]_{\gamma;\varepsilon}} [[e_2[R \uplus \Delta R]]]_{\gamma;\varepsilon[x:=v]} \\
&= \Big[\biguplus_{v \in [[e_1[R]]]_{\gamma;\varepsilon}} [[e_2[R \uplus \Delta R]]]_{\gamma;\varepsilon[x:=v]}\Big] \uplus \Big[\biguplus_{v \in [[\delta(e_1)[R,\Delta R]]]_{\gamma;\varepsilon}} [[e_2[R \uplus \Delta R]]]_{\gamma;\varepsilon[x:=v]}\Big] \\
&= \Big[\biguplus_{v \in [[e_1[R]]]_{\gamma;\varepsilon}} [[e_2[R]]]_{\gamma;\varepsilon[x:=v]} \uplus [[\delta(e_2)[R,\Delta R]]]_{\gamma;\varepsilon[x:=v]}\Big] \uplus \\
&\quad \Big[\biguplus_{v \in [[\delta(e_1)[R,\Delta R]]]_{\gamma;\varepsilon}} [[e_2[R]]]_{\gamma;\varepsilon[x:=v]} \uplus [[\delta(e_2)[R,\Delta R]]]_{\gamma;\varepsilon[x:=v]}\Big] \\
&= \Big[\biguplus_{v \in [[e_1[R]]]_{\gamma;\varepsilon}} [[e_2[R]]]_{\gamma;\varepsilon[x:=v]}\Big] \uplus \Big[\biguplus_{v \in [[e_1[R]]]_{\gamma;\varepsilon}} [[\delta(e_2)[R,\Delta R]]]_{\gamma;\varepsilon[x:=v]}\Big] \uplus \\
&\quad \Big[\biguplus_{v \in [[\delta(e_1)[R,\Delta R]]]_{\gamma;\varepsilon}} [[e_2[R]]]_{\gamma;\varepsilon[x:=v]}\Big] \uplus \Big[\biguplus_{v \in [[\delta(e_1)[R,\Delta R]]]_{\gamma;\varepsilon}} [[\delta(e_2)[R,\Delta R]]]_{\gamma;\varepsilon[x:=v]}\Big]
\end{aligned}
$$

$$= [[(\textbf{for } x \textbf{ in } e_1 \textbf{ union } e_2)[R]]]_{\gamma;\varepsilon} \uplus [[(\textbf{for } x \textbf{ in } e_1 \textbf{ union } \delta(e_2))[R,\Delta R]]]_{\gamma;\varepsilon} \uplus$$
$$[[(\textbf{for } x \textbf{ in } \delta(e_1) \textbf{ union } e_2)[R,\Delta R]]]_{\gamma;\varepsilon} \uplus [[(\textbf{for } x \textbf{ in } \delta(e_1) \textbf{ union } \delta(e_2))[R,\Delta R]]]_{\gamma;\varepsilon}$$
$$= [[(\textbf{for } x \textbf{ in } e_1 \textbf{ union } e_2)[R]]]_{\gamma;\varepsilon} \uplus [[\delta(\textbf{for } x \textbf{ in } e_1 \textbf{ union } e_2)[R,\Delta R]]]_{\gamma;\varepsilon}$$
$$= [[(\textbf{for } x \textbf{ in } e_1 \textbf{ union } e_2)[R] \uplus \delta(\textbf{for } x \textbf{ in } e_1 \textbf{ union } e_2)[R,\Delta R]]]_{\gamma;\varepsilon}$$

- For $h = e_1 \times e_2$ the reasoning is similar to the case of $h = \textbf{for } x \textbf{ in } e_1 \textbf{ union } e_2$.
- For $h = e_1 \uplus e_2$ the result follows from the associativity and commutativity of $\uplus$.
- For $h = \ominus(e)$ the result follows from the associativity and commutativity of $\uplus$ and the fact that $\ominus$ is the inverse operation wrt. $\uplus$.
- For $h = \textbf{flatten}(e)$:

$$[[\textbf{flatten}(e)[R \uplus \Delta R]]] = \biguplus_{v \in [[e[R\uplus\Delta R]]]} v = \biguplus_{v \in [[e[R]]] \uplus [[\delta(e)[R,\Delta R]]]} v =$$
$$= \biguplus_{v \in [[e[R]]]} v \quad \uplus \quad \biguplus_{v \in [[\delta(e)[R,\Delta R]]]} v = [[\textbf{flatten}(e)[R]]] \uplus [[\textbf{flatten}(\delta(e))[R,\Delta R]]] =$$
$$= [[\textbf{flatten}(e)[R] \uplus \textbf{flatten}(\delta(e))[R,\Delta R]]] = [[\textbf{flatten}(e)[R] \uplus \delta(\textbf{flatten}(e))[R,\Delta R]]]$$

- For $h = \textbf{let } X := e_1 \textbf{ in } e_2$

$$[[(\textbf{let } X := e_1 \textbf{ in } e_2)[R \uplus \Delta R]]]_{\gamma;\varepsilon} = [[e_2[R \uplus \Delta R, X]]]_{\gamma;\varepsilon[X:=[[e_1[R\uplus\Delta R]]]_{\gamma;\varepsilon}]} =$$
$$= [[e_2[R,X]]]_{\gamma;\varepsilon[X:=[[e_1[R\uplus\Delta R]]]_{\gamma;\varepsilon}]} \uplus [[\delta_R(e_2)[R,X,\Delta R]]]_{\gamma;\varepsilon[X:=[[e_1[R\uplus\Delta R]]]_{\gamma;\varepsilon}]}$$
$$= [[e_2[R,X]]]_{\gamma;\varepsilon[X:=[[e_1[R]]]_{\gamma;\varepsilon}\uplus[[\delta(e_1)[R,\Delta R]]]_{\gamma;\varepsilon}]} \quad \uplus$$
$$[[\delta_R(e_2)[R,X,\Delta R]]]_{\gamma;\varepsilon[X:=[[e_1[R]]]_{\gamma;\varepsilon}\uplus[[\delta(e_1)[R,\Delta R]]]_{\gamma;\varepsilon}]}$$
$$= [[e_2[R, X \uplus \Delta X]]]_{\gamma;\varepsilon[X:=[[e_1[R]]]_{\gamma;\varepsilon},\Delta X:=[[\delta(e_1)[R,\Delta R]]]_{\gamma;\varepsilon}]} \quad \uplus$$
$$[[\delta_R(e_2)[R, X \uplus \Delta X, \Delta R]]]_{\gamma;\varepsilon[X:=[[e_1[R]]]_{\gamma;\varepsilon},\Delta X:=[[\delta(e_1)[R,\Delta R]]]_{\gamma;\varepsilon}]}$$
$$= [[e_2[R, X \uplus \Delta X] \uplus \delta_R(e_2)[R, X \uplus \Delta X, \Delta R]]]_{\gamma;\varepsilon[X:=[[e_1[R]]]_{\gamma;\varepsilon},\Delta X:=[[\delta(e_1)[R,\Delta R]]]_{\gamma;\varepsilon}]}$$
$$= [[e_2[R,X] \uplus \delta_X(e_2)[R,X,\Delta X] \uplus \delta_R(e_2)[R,X,\Delta R] \uplus$$
$$\delta_X(\delta_R(e_2))[R,X,\Delta X,\Delta R]]]_{\gamma;\varepsilon[X:=[[e_1[R]]]_{\gamma;\varepsilon},\Delta X:=[[\delta(e_1)[R,\Delta R]]]_{\gamma;\varepsilon}]}$$
$$= \textbf{let } X := e_1[R], \Delta X := \delta(e_1)[R,\Delta R] \textbf{ in}$$
$$(e_2[R,X] \uplus \delta_X(e_2)[R,X,\Delta X] \uplus \delta_R(e_2)[R,X,\Delta R] \uplus \delta_X(\delta_R(e_2))[R,X,\Delta X,\Delta R])$$

□

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

Proof. We do a case by case analysis on h.

- For $h \in \{\emptyset, p, \mathbf{sng}(\langle\rangle), \mathbf{sng}(x), \mathbf{sng}(\pi_i(x)), \mathbf{sng}^*(e)\}$ we have from the definition of $\delta(\cdot)$ that $\delta(h) = \emptyset$.
- For $h = \mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2$, we have by the induction hypothesis that $\delta(e_1) = \emptyset$, $\delta(e_2) = \emptyset$, therefore $\delta(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2) = (\mathbf{for}\ x\ \mathbf{in}\ \emptyset\ \mathbf{union}\ e_2) \uplus (\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ \emptyset) \uplus (\mathbf{for}\ x\ \mathbf{in}\ \emptyset\ \mathbf{union}\ \emptyset) = \emptyset$.
- For $h = e_1 \times e_2$ the reasoning is similar to the case of $h = \mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2$.
- For $h = e_1 \uplus e_2$, we have by the induction hypothesis that $\delta(e_1) = \emptyset$, $\delta(e_2) = \emptyset$, therefore $\delta(e_1 \uplus e_2) = \emptyset \uplus \emptyset = \emptyset$.
- For $h = \ominus(e)$, we have by the induction hypothesis that $\delta(e) = \emptyset$, therefore $\delta(\ominus(e)) = \ominus(\emptyset) = \emptyset$.
- For $h = \mathbf{flatten}(e)$, we have by the induction hypothesis that $\delta(e) = \emptyset$, therefore $\delta(\mathbf{flatten}(e)) = \mathbf{flatten}(\emptyset) = \emptyset$.
- For $h = \mathbf{let}\ X := e_1\ \mathbf{in}\ e_2$, we have by the induction hypothesis that $\delta_R(e_2) = \emptyset$, $\Delta X = \delta_R(e_1) = \emptyset$, and the result follows from the fact the $\delta_X(e_2)[X, \emptyset] = \emptyset$.

□

A.1.2 The cost transformation

Lemma 15. *For any* IncNRC$^+$ *expression* $\Gamma; \Pi, x : C \vdash h : \mathbf{Bag}(A)$*, the cost interpretation* $\mathcal{C}[[h]]$ *is monotonic, i.e.* $\forall c_1, c_2 \in C^\circ$ *s.t.* $c_1 \preceq c_2$ *then* $\mathcal{C}[[h]]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \preceq \mathcal{C}[[h]]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}$.

Proof. The result follows via structural induction on h and from the fact that the cost functions of the IncNRC$^+$ constructs are themselves monotonic.

We do a case by case analysis on h:

- For $h \in \{R, p, \emptyset, \mathbf{sng}(\langle\rangle)\}$ the result follows from the fact that $\forall c_1, c_2.\ \mathcal{C}[[h]]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} = \mathcal{C}[[h]]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}$.
- For $h = \mathbf{sng}(x) : \mathcal{C}[[\mathbf{sng}(x)]]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} = \{c_1\} \preceq \{c_2\} = \mathcal{C}[[\mathbf{sng}(x)]]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}$
- For $h = \mathbf{sng}(\pi_x(i)) : \mathcal{C}[[\mathbf{sng}(\pi_x(i))]]_{\gamma^\circ;\varepsilon^\circ[x:=\langle c_{11}, c_{12}\rangle]} = \{c_{1i}\} \preceq \{c_{2i}\} = \mathcal{C}[[\mathbf{sng}(\pi_x(i))]]_{\gamma^\circ;\varepsilon^\circ[x:=\langle c_{21}, c_{22}\rangle]}$
- For $h = \mathbf{for}\ y\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2$, we have from the induction hypothesis that:

$$\mathcal{C}_i[[e_1]]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \preceq \mathcal{C}_i[[e_1]]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}$$
$$\mathcal{C}_o[[e_1]]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \preceq \mathcal{C}_o[[e_1]]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}$$
$$\mathcal{C}_i[[e_2]]_{\gamma^\circ;\varepsilon^\circ[y:=\mathcal{C}_i[[e_1]]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]}]} \preceq \mathcal{C}_i[[e_2]]_{\gamma^\circ;\varepsilon^\circ[y:=\mathcal{C}_i[[e_1]]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}]}$$

$$\mathcal{C}_o[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[y:=\mathcal{C}_i[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]}]} \leq \mathcal{C}_o[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[y:=\mathcal{C}_i[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}]},$$

therefore:

$$\begin{aligned}
&\mathcal{C}[\![\textbf{for } x \textbf{ in } e_1 \textbf{ union } e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} = \\
&= \mathcal{C}_o[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[y:=\mathcal{C}_i[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]}]} \cdot \mathcal{C}_o[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]}\{\mathcal{C}_i[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[y:=\mathcal{C}_i[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]}]}\} \\
&\leq \mathcal{C}_o[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[y:=\mathcal{C}_i[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}]} \cdot \mathcal{C}_o[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}\{\mathcal{C}_i[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[y:=\mathcal{C}_i[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}]}\} \\
&= \mathcal{C}[\![\textbf{for } x \textbf{ in } e_1 \textbf{ union } e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}
\end{aligned}$$

- For $h = e_1 \times e_2$, we have from the induction hypothesis that

$$\begin{aligned}
&\mathcal{C}_i[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \leq \mathcal{C}_i[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]} &\qquad \mathcal{C}_i[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \leq \mathcal{C}_i[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]} \\
&\mathcal{C}_o[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \leq \mathcal{C}_o[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]} &\qquad \mathcal{C}_o[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \leq \mathcal{C}_o[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]},
\end{aligned}$$

therefore:

$$\begin{aligned}
&\mathcal{C}[\![e_1 \times e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} = \\
&= \mathcal{C}_o[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \cdot \mathcal{C}_o[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]}\{\langle \mathcal{C}_i[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]}, \mathcal{C}_i[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]}\rangle\} \\
&\leq \mathcal{C}_o[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]} \cdot \mathcal{C}_o[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}\{\langle \mathcal{C}_i[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}, \mathcal{C}_i[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}\rangle\} \\
&= \mathcal{C}[\![e_1 \times e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}
\end{aligned}$$

- For $h = e_1 \uplus e_2$, we have from the induction hypothesis that $\mathcal{C}[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \leq \mathcal{C}[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}$ and $\mathcal{C}[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \leq \mathcal{C}[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}$, therefore:

$$\begin{aligned}
&\mathcal{C}[\![e_1 \uplus e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} = \sup(\mathcal{C}[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]}, \mathcal{C}[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]}) \\
&\quad \leq \sup(\mathcal{C}[\![e_1]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}, \mathcal{C}[\![e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}) = \mathcal{C}[\![e_1 \uplus e_2]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}
\end{aligned}$$

- For $h = \ominus(e)$, we have from the induction hypothesis that $\mathcal{C}[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \leq \mathcal{C}[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}$, therefore:

$$\mathcal{C}[\![\ominus(e)]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} = \mathcal{C}[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \leq \mathcal{C}[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]} = \mathcal{C}[\![\ominus(e)]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}$$

- For $h = \textbf{flatten}(e)$, we have from the induction hypothesis that $\mathcal{C}_o[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \leq \mathcal{C}_o[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}$, $\mathcal{C}_{io}[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \leq \mathcal{C}_{io}[\![f]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}$ and $\mathcal{C}_{ii}[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \leq \mathcal{C}_{ii}[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}$, therefore:

$$\begin{aligned}
&\mathcal{C}[\![\textbf{flatten}(e)]\!](c_1) = \mathcal{C}_o[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \cdot \mathcal{C}_{io}[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]}\{\mathcal{C}_{ii}[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]}\} \\
&\quad \leq \mathcal{C}_o[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]} \cdot \mathcal{C}_{io}[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}\{\mathcal{C}_{ii}[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}\} = \mathcal{C}[\![\textbf{flatten}(e)]\!](c_2)
\end{aligned}$$

- For $h = \textbf{sng}^*(e)$, we have from the induction hypothesis that $\mathcal{C}[\![e]\!]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} \leq$

$\mathcal{C}[[f]]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}$, therefore:

$$\mathcal{C}[[\mathbf{sng}^*(e)]]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]} = \{\mathcal{C}[[e]]_{\gamma^\circ;\varepsilon^\circ[x:=c_1]}\} \preceq \{\mathcal{C}[[e]]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}\} = \mathcal{C}[[\mathbf{sng}^*(e)]]_{\gamma^\circ;\varepsilon^\circ[x:=c_2]}$$

□

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

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

Proof. The proof follows via structural induction on h and from the cost semantics of IncNRC$^+$ constructs as well as the monotonicity of tcost$(\cdot)$.

- For $h = R$ we have: $\mathcal{C}[[\delta(R)]] = \mathcal{C}[[\Delta R]] = \text{size}(\Delta R) \prec \text{size}(R) = \mathcal{C}[[R]]$
- For $h =$ **for** x **in** e_1 **union** e_2 we need to show that:

$$\begin{aligned}
&\mathcal{C}[[\delta(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2)]] = \\
&= \mathcal{C}[[(\mathbf{for}\ x\ \mathbf{in}\ \delta(e_1)\ \mathbf{union}\ e_2) \uplus (\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ \delta(e_2)) \uplus \\
&\qquad (\mathbf{for}\ x\ \mathbf{in}\ \delta(e_1)\ \mathbf{union}\ \delta(e_2))]] \\
&= \sup(\mathcal{C}[[\mathbf{for}\ x\ \mathbf{in}\ \delta(e_1)\ \mathbf{union}\ e_2]], \mathcal{C}[[\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ \delta(e_2)]], \\
&\qquad\quad \mathcal{C}[[\mathbf{for}\ x\ \mathbf{in}\ \delta(e_1)\ \mathbf{union}\ \delta(e_2)]]) \\
&\prec \mathcal{C}[[\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2]]
\end{aligned}$$

Case 1: $\mathcal{C}[[\delta(e_1)]] \prec \mathcal{C}[[e_1]]$ and e_2 is *input-independent*, therefore $\delta(e_2) = \emptyset$ (from Lemma 1):

$$\begin{aligned}
&\mathcal{C}[[\delta(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2)]] = \mathcal{C}[[\mathbf{for}\ x\ \mathbf{in}\ \delta(e_1)\ \mathbf{union}\ e_2]] = \\
&= \mathcal{C}_o[[e_2]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[\delta(e_1)]]]} \cdot \mathcal{C}_o[[\delta(e_1)]]\{\mathcal{C}_i[[e_2]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[\delta(e_1)]]]}\} \\
&\prec \mathcal{C}_o[[e_2]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[e_1]]]} \cdot \mathcal{C}_o[[e_1]]\{\mathcal{C}_i[[e_2]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[e_1]]]}\} = \mathcal{C}[[\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2]],
\end{aligned}$$

where we used the fact that $\mathcal{C}_o[[\delta(e_1)]] \prec \mathcal{C}_o[[e_1]]$ and $\mathcal{C}[[e_2]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[\delta(e_1)]]]} \preceq \mathcal{C}[[e_2]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[e_1]]]}$ (from Lemma 15).

Case 2: $\mathcal{C}[[\delta(e_2)]] \prec \mathcal{C}[[e_2]]$ and e_1 is *input-independent*, therefore $\delta(e_1) = \emptyset$ (from Lemma 1):

$$\begin{aligned}
&\mathcal{C}[[\delta(\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2)]] = \mathcal{C}[[\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ \delta(e_2)]] \\
&= \mathcal{C}_o[[\delta(e_2)]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[e_1]]]} \cdot \mathcal{C}_o[[e_1]]\{\mathcal{C}_i[[\delta(e_2)]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[e_1]]]}\} \\
&\prec \mathcal{C}_o[[e_2]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[e_1]]]} \cdot \mathcal{C}_o[[e_1]]\{\mathcal{C}_i[[e_2]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[e_1]]]}\} = \mathcal{C}[[\mathbf{for}\ x\ \mathbf{in}\ e_1\ \mathbf{union}\ e_2]],
\end{aligned}$$

where we used the fact that $\mathcal{C}_i[[\delta(e_2)]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[e_1]]]} \preceq \mathcal{C}_i[[e_2]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[e_1]]]}$ and $\mathcal{C}_o[[\delta(e_2)]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[e_1]]]} \prec \mathcal{C}_o[[e_2]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[e_1]]]}$.

Case 3: $\mathcal{C}[[\delta(e_2)]] \prec \mathcal{C}[[e_2]]$ and $\mathcal{C}[[\delta(e_1)]] \prec \mathcal{C}[[e_1]]$. We show that each term of the sup function is less than the cost of the original function:

$\mathcal{C}[[\textbf{for } x \textbf{ in } \delta(e_1) \textbf{ union } e_2]] \prec \mathcal{C}[[\textbf{for } x \textbf{ in } e_1 \textbf{ union } e_2]]$, see Case 1.

$\mathcal{C}[[\textbf{for } x \textbf{ in } e_1 \textbf{ union } \delta(e_2)]] \prec \mathcal{C}[[\textbf{for } x \textbf{ in } e_1 \textbf{ union } e_2]]$, see Case 2.

$$\begin{aligned}
&\mathcal{C}[[\textbf{for } x \textbf{ in } \delta(e_1) \textbf{ union } \delta(e_2)]](c) = \\
&= \mathcal{C}_o[[\delta(e_2)]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[\delta(e_1)]]]} \cdot \mathcal{C}_o[[\delta(e_1)]]\{\mathcal{C}_i[[\delta(e_2)]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[\delta(e_1)]]]}\} \\
&\prec \mathcal{C}_o[[e_2]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[\delta(e_1)]]]} \cdot \mathcal{C}_o[[\delta(e_1)]]\{\mathcal{C}_i[[e_2]]_{\gamma^\circ;\varepsilon^\circ[x:=\mathcal{C}_i[[\delta(e_1)]]]}\} \\
&= \mathcal{C}[[\textbf{for } x \textbf{ in } \delta(e_1) \textbf{ union } e_2]] \prec \mathcal{C}[[\textbf{for } x \textbf{ in } e_1 \textbf{ union } e_2]].
\end{aligned}$$

- For $h = e_1 \times e_2$ we need to show that:

$$\begin{aligned}
\mathcal{C}[[\delta(e_1 \times e_2)]] &= \mathcal{C}[[e_1 \times \delta(e_2) \uplus \delta(e_1) \times e_2 \uplus \delta(e_1) \times \delta(e_2)]] \\
&= \sup(\mathcal{C}[[e_1 \times \delta(e_2)]], \mathcal{C}[[\delta(e_1) \times e_2]], \mathcal{C}[[\delta(e_1) \times \delta(e_2)]]) \prec \mathcal{C}[[e_1 \times e_2]]
\end{aligned}$$

Case 1: $\mathcal{C}[[\delta(e_2)]] \prec \mathcal{C}[[e_2]]$ and e_1 is *input-independent*, therefore $\delta(e_1) = \varnothing$ (from Lemma 1):

$$\begin{aligned}
\mathcal{C}[[\delta(e_1 \times e_2)]] = \mathcal{C}[[e_1 \times \delta(e_2)]] &= \mathcal{C}_o[[e_1]] \cdot \mathcal{C}_o[[\delta(e_2)]]\{\langle \mathcal{C}_i[[e_1]], \mathcal{C}_i[[\delta(e_2)]]\rangle\} \\
&\prec \mathcal{C}_o[[e_1]] \cdot \mathcal{C}_o[[e_2]]\{\langle \mathcal{C}_i[[e_1]], \mathcal{C}_i[[e_2]]\rangle\} = \mathcal{C}[[e_1 \times e_2]]
\end{aligned}$$

Case 2: $\mathcal{C}[[\delta(e_1)]] \prec \mathcal{C}[[e_1]]$ and e_2 is *input-independent*: Analogous to Case 1.

Case 3: $\mathcal{C}[[\delta(e_1)]] \prec \mathcal{C}[[e_1]]$ and $\mathcal{C}[[\delta(e_2)]] \prec \mathcal{C}[[e_2]]$. We show that each term of the sup function is less than the cost of the original function:

$\mathcal{C}[[e_1 \times \delta(e_2)]] \prec \mathcal{C}[[e_1 \times e_2]]$, see Case 2.

$\mathcal{C}[[\delta(e_1) \times e_2]] \prec \mathcal{C}[[e_1 \times e_2]]$, see Case 3.

$$\begin{aligned}
\mathcal{C}[[\delta(e_1) \times \delta(e_2)]] &= \mathcal{C}_o[[\delta(e_1)]] \cdot \mathcal{C}_o[[\delta(e_2)]]\{\langle \mathcal{C}_i[[\delta(e_1)]], \mathcal{C}_i[[\delta(e_2)]]\rangle\} \\
&\prec \mathcal{C}_o[[e_1]] \cdot \mathcal{C}_o[[\delta(e_2)]]\{\langle \mathcal{C}_i[[e_1]], \mathcal{C}_i[[\delta(e_2)]]\rangle\} = \mathcal{C}[[e_1 \times \delta(e_2)]] \prec \mathcal{C}[[e_1 \times e_2]].
\end{aligned}$$

- For $h = e_1 \uplus e_2$ we have the following cases:
  Case 1: $\mathcal{C}[[\delta(e_2)]] \prec \mathcal{C}[[e_2]]$ and e_1 is *input-independent*, therefore $\delta(e_1) = \varnothing$ (from Lemma 1):

$$\mathcal{C}[[\delta(e_1 \uplus e_2)]] = \mathcal{C}[[\delta(e_2)]] \prec \sup(\mathcal{C}[[e_1]], \mathcal{C}[[e_2]]) = \mathcal{C}[[e_1 \uplus e_2]].$$

Case 2: $\mathcal{C}[[\delta(e_1)]] \prec \mathcal{C}[[e_1]]$ and e_2 is *input-independent*: Analogous to Case 1.

Case 3: $\mathcal{C}[[\delta(e_1)]] \prec \mathcal{C}[[e_1]]$ and $\mathcal{C}[[\delta(e_2)]] \prec \mathcal{C}[[e_2]]$:

$$\begin{aligned}\mathcal{C}[[\delta(e_1 \uplus e_2)]] = \mathcal{C}[[\delta(e_1) \uplus \delta(e_2)]] &= \sup(\mathcal{C}[[\delta(e_1)]], \mathcal{C}[[\delta(e_2)]]) \\ &\prec \sup(\mathcal{C}[[e_1]], \mathcal{C}[[e_2]]) = \mathcal{C}[[e_1 \uplus e_2]].\end{aligned}$$

- For $h = \ominus(e)$ we have that $\mathcal{C}[[\delta(\ominus(e))]] = \mathcal{C}[[\ominus(\delta(e))]] = \mathcal{C}[[\delta(e)]] \prec \mathcal{C}[[e]] = \mathcal{C}[[\ominus(e)]]$.
- For $h = \textbf{flatten}(e)$ we have that $\mathcal{C}[[\delta(e)]] \prec \mathcal{C}[[e]]$, therefore:

$$\begin{aligned}\mathcal{C}[[\delta(\textbf{flatten}(e))]] = \mathcal{C}[[\textbf{flatten}(\delta(e))]] &= \mathcal{C}_o[[\delta(e)]] \cdot \mathcal{C}_{oi}[[\delta(e)]]\{\mathcal{C}_{ii}[[\delta(e)]]\} \\ &\prec \mathcal{C}_o[[e]] \cdot \mathcal{C}_{oi}[[e]]\{\mathcal{C}_{ii}[[e]]\} = \mathcal{C}[[\textbf{flatten}(e)]],\end{aligned}$$

where we used the fact that $\mathcal{C}_o[[\delta(e)]] \prec \mathcal{C}_o[[e]]$ and $\mathcal{C}_i[[\delta(e)]] \preceq \mathcal{C}_i[[e]]$.

□

A.2 Consistency of shredded values

Given an input bag $R : \textbf{Bag}(A)$, its shredding version consists of a flat component $R^F : \textbf{Bag}(A^F)$ and a context component $R^\Gamma : A^\Gamma$, which is essentially a tuple of dictionaries $d_k : \mathbb{L} \to \textbf{Bag}(C^F)$ such that the definition of any label l in d_k corresponds to a inner bag of type $\textbf{Bag}(C)$ from R.

In order to be able to manipulate shredded values in a consistent manner we must guarantee that i) the union of label dictionaries is always well defined and that ii) each label occurring in the flat component of a shredded value has a corresponding definition in the associated context component. More formally:

Definition 4. *We say that shredded bags $\langle R_i^F, R_i^\Gamma \rangle : \textbf{Bag}(A_i^F) \times A_i^\Gamma$ are consistent if the union operation over dictionaries is well-defined between any two compatible dictionaries in $R_i^\Gamma, 1 \leq i \leq n$ and if all the elements of R_i^F are* consistent *wrt. their context R_i^Γ, where $v : A^F$ is* consistent *wrt. $v^\Gamma : A^\Gamma$, if:*

- *$A = Base$ or*
- *$A = A_1 \times A_2$, $v = \langle v_1, v_2 \rangle$, $v^\Gamma = \langle v_1^\Gamma, v_2^\Gamma \rangle$ and v_1, v_2 are* consistent *wrt. v_1^Γ and v_2^Γ, respectively, or*
- *$A = \textbf{Bag}(C)$, $v = l : \mathbb{L}$, $v^\Gamma = \langle v^D, c^\Gamma \rangle : (\mathbb{L} \to \textbf{Bag}(C^F)) \times C^\Gamma$, there exists a definition for l in v^D (i.e. $l \in \operatorname{supp}(v^D)$) and for every element c_j of the definition $v^D(l) = \biguplus_j \{c_j\}$, c_j is* consistent *wrt. c^Γ.*

Regarding the first requirement, we note that the union of label dictionaries $d_1 \cup d_2$ results in an error when a label l is defined in both d_1 and d_2 (i.e. $l \in \operatorname{supp}(d_1) \cap \operatorname{supp}(d_2)$) but the definitions do not match. Therefore, in order to avoid this scenario a label l must have the same definitions in all dictionaries where it appears. This is true for shredded input bags, since the shredding function introduces a fresh label for every inner bag encountered in the process. Furthermore, this property continues to be true after evaluating the shredding of query $h[R_i] : \mathbf{Bag}(B)$:

$$\mathrm{sh}^F(h)[R_i^F, R_i^\Gamma] : \mathbf{Bag}(B^F) \qquad\qquad \mathrm{sh}^\Gamma(h)[R_i^F, R_i^\Gamma] : B^\Gamma$$

over shredded input bags $R_i^F : \mathbf{Bag}(A^F), R_i^\Gamma : A^\Gamma$ because a) the labels introduced by the query (corresponding to the shredding of $\mathbf{sng}(f)$ constructs) are guaranteed to be fresh and b) within the queries $\mathrm{sh}^F(h)$ and $\mathrm{sh}^\Gamma(h)$ dictionaries are combined only via label union which doesn't modify label definitions (i.e. we never apply bag union over dictionaries).

Lemma 16. *Shredding produces consistent values, i.e. for any input bags R_i, their shredding $R_i^F = \mathbf{for}\ r\ \mathbf{in}\ R_i\ \mathbf{union}\ \mathrm{s}^F_{A_i}(r), R_i^\Gamma = \mathrm{s}^\Gamma_{A_i}$ is consistent.*

Lemma 17. *Shredded* NRC$^+$ *queries preserve consistency of shredded bags, i.e. for any* NRC$^+$ *query $h[R_i]$, the output of $\langle h^F, h^\Gamma\rangle[R_i^F, R_i^\Gamma]$ over consistent shredded bags $\langle R_i^F, R_i^\Gamma\rangle$, is also consistent.*

When discussing the update of shredded bags $\langle R_i^F, R_i^\Gamma\rangle$ by pointwise bag union with $\langle \Delta R_i^F, \Delta R_i^\Gamma\rangle$ we require that both shreddings are independently consistent. Nonetheless, the definition of a label l from R_i^Γ will most likely differ from its definition in ΔR_i^Γ since the first one contains the initial value of the bag denoted by l, while the second one represents its update. We remark that this does not create a problem wrt. label union of dictionaries since we only union two dictionaries which are both from R_i^Γ or ΔR_i^Γ, but we never label union a dictionary from R_i^Γ with a dictionary from ΔR_i^Γ.

The definitions provided by ΔR_i^Γ can be split in two categories: i) update definitions for bags that have been initially defined in R_i^Γ; and ii) fresh definitions corresponding to new labels introduced in the delta update. We require that if a label $l \in \mathrm{supp}(R_i^\Gamma)$ has an update definition in ΔR_i^Γ, then that label must have the same update definition in every $\Delta R_k^\Gamma, k = 1..n$, for which $l \in \mathrm{supp}(R_k^\Gamma)$. This is necessary in order to make sure that the resulting shredded value $\langle R_i^F \uplus \Delta R_i^F, R_i^\Gamma \uplus \Delta R_i^\Gamma\rangle$ is also consistent. For the fresh definitions we require that their labels are distinct from any label introduced by $R_k^\Gamma, k = 1..n$. More formally:

Definition 5. *We say that update $\langle \Delta R_i^F, \Delta R_i^\Gamma\rangle$ is consistent wrt. shredded bags $\langle R_i^F, R_i^\Gamma\rangle$ if both $\langle \Delta R_i^F, \Delta R_i^\Gamma\rangle$ and $\langle R_i^F, R_i^\Gamma\rangle$ are consistent and*

- *for every label $l \in \mathrm{supp}(\Delta R_i^\Gamma) \cap \mathrm{supp}(R_i^\Gamma) \cap \mathrm{supp}(R_k^\Gamma)$ then $l \in \mathrm{supp}(\Delta R_k^\Gamma), k = 1..n$.*
- *for every label $l \in \mathrm{supp}(\Delta R_i^\Gamma) \setminus \mathrm{supp}(R_i^\Gamma)$ then $l \notin \mathrm{supp}(R_k^\Gamma), k = 1..n$.*

Lemma 18. *Deltas of shredded* NRC$^+$ *queries preserve consistency of updates, i.e. for any* NRC$^+$ *query $h[R_i]$ and shredded update $\langle \Delta R_i^F, \Delta R_i^\Gamma\rangle$ consistent wrt. shredded input $\langle R_i^F, R_i^\Gamma\rangle$,* *then the output update* $\langle\delta(h^F),\delta(h^\Gamma)\rangle[R_i^F,R_i^\Gamma,\Delta R_i^F,\Delta R_i^\Gamma]$ *is also consistent wrt. output* $\langle h^F,h^\Gamma\rangle[R_i^F,R_i^\Gamma]$.

Proof. The first requirement of Definition 5 follows from the fact that if $l\in\text{supp}(\delta(h_j^\Gamma))\cap\text{supp}(h_j^\Gamma)\cap\text{supp}(h_k^\Gamma)$, where h_j^Γ/h_k^Γ stands for the j'th/k'th dictionary in h^Γ, then taking delta over h_k^Γ will also produce a definition for l in $\delta(h_k^\Gamma)$.

As the delta transformation does not add any new labels we have that:

$$\text{supp}(h_j^\Gamma)\subseteq\text{supp}_h\cup\text{supp}(R_i^\Gamma)\qquad\text{supp}(\delta(h_j^\Gamma))\subseteq\text{supp}_h\cup\text{supp}(R_i^\Gamma)\cup\text{supp}(\Delta R_i^\Gamma),$$

where supp_h represents the labels introduced by the query h itself via singleton constructs $\mathbf{sng}_\iota(e)$.

For the second requirement of Definition 5 we note that if $l\in\text{supp}(\delta(h_j^\Gamma))\setminus\text{supp}(h_j^\Gamma)$, then $l\in\text{supp}(\Delta R_i^\Gamma)\setminus R_i^\Gamma$. Therefore, $l\notin\text{supp}(h_k^\Gamma)$ for any dictionary in h^Γ. □

A.3 Delta transformation for IncNRC_l^+

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:*

$$\begin{aligned}&h[R\uplus\Delta R]=h[R]\uplus\delta(h)[R,\Delta R],\\&\deg(\delta(h))=\deg(h)-1\qquad\textit{and}\\&\text{tcost}(\mathcal{C}[[\delta(h)]])<\text{tcost}(\mathcal{C}[[h]]).\end{aligned}$$

Proof. The proof follows by structural induction on h and from the semantics of IncNRC_l^+ constructs.

- For $h=\text{inL}_l$ we have $\delta(h)=\varnothing$ as the query does not depend on the input bags R_i and the result follows immediately.
- For $h=[(\iota,\Pi)\mapsto e](l')=$ **for** $\langle\iota',\varepsilon\rangle$ **in sng**(l') **where** $\iota==\iota'$ **union** $\rho_\varepsilon(e)$, the result follows from the delta of **for** and from the fact that **sng**(l') does not depend on the input bags, therefore its delta is $\varnothing$.
- For $h=e_1\cup e_2,e_1,e_2:\mathbb{L}\to\mathbf{Bag}(A)$, we need to show that for any $l\in\mathbb{L}$:

  $$[[(e_1^{\text{old}}\uplus e_1^\Delta)\cup(e_2^{\text{old}}\uplus e_2^\Delta)]](l)=[[(e_1^{\text{old}}\cup e_2^{\text{old}})\uplus(e_1^\Delta\cup e_2^\Delta)]](l),$$

  where: $e_k^{\text{old}}=e_k[R_i^F,R_i^\Gamma]$, and $e_k^\Delta=\delta(e_k)[R_i^F,R_i^\Gamma,\Delta R_i^F,\Delta R_i^\Gamma]$, with $k=1,2$.

  We assume that update $\langle\Delta R_i^F,\Delta R_i^\Gamma\rangle$ is consistent wrt. input bags $\langle R_i^F,R_i^\Gamma\rangle$ and from Lemma 18 we conclude that update $\langle e_1^\Delta,e_2^\Delta\rangle$ is also consistent wrt. $\langle e_1^{\text{old}},e_2^{\text{old}}\rangle$.

We do a case analysis on l (there are 16 possibilities):

- $l \notin \mathrm{supp}(e_1^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_1^{\Delta})$, $l \notin \mathrm{supp}(e_2^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_2^{\Delta})$. Trivial.
- $l \in \mathrm{supp}(e_1^{\mathrm{old}})$, $l \in \mathrm{supp}(e_1^{\Delta})$, $l \in \mathrm{supp}(e_2^{\mathrm{old}})$, $l \in \mathrm{supp}(e_2^{\Delta})$. From the consistency of $\langle e_1^{\mathrm{old}}, e_2^{\mathrm{old}}\rangle$ we have that $e_1^{\mathrm{old}}(l) = e_2^{\mathrm{old}}(l)$. Similarly, we get that $e_1^{\Delta}(l) = e_2^{\Delta}(l)$. Therefore, we have that: $(e_1^{\mathrm{old}} \uplus e_1^{\Delta})(l) = (e_2^{\mathrm{old}} \uplus e_2^{\Delta})(l)$ and $((e_1^{\mathrm{old}} \uplus e_1^{\Delta}) \cup (e_2^{\mathrm{old}} \uplus e_2^{\Delta}))(l) = (e_1^{\mathrm{old}} \uplus e_1^{\Delta})(l) = ((e_1^{\mathrm{old}} \cup e_2^{\mathrm{old}}) \uplus (e_1^{\Delta} \cup e_2^{\Delta}))(l)$
- Two cases lead to a contradiction with the first requirement of a consistent update value, since the label l is defined in both e_1^{old} and e_2^{old}, but is updated by only one of $e_1^{\Delta}/e_2^{\Delta}$.
  * $l \in \mathrm{supp}(e_1^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_1^{\Delta})$, $l \in \mathrm{supp}(e_2^{\mathrm{old}})$, $l \in \mathrm{supp}(e_2^{\Delta})$.
  * $l \in \mathrm{supp}(e_1^{\mathrm{old}})$, $l \in \mathrm{supp}(e_1^{\Delta})$, $l \in \mathrm{supp}(e_2^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_2^{\Delta})$.
- Four cases lead to a contradiction with the second requirement of a consistent update value since $e_1^{\Delta}/e_2^{\Delta}$ introduce a fresh definition for a label that already appears in $e_2^{\mathrm{old}}/e_1^{\mathrm{old}}$.
  * $l \notin \mathrm{supp}(e_1^{\mathrm{old}})$, $l \in \mathrm{supp}(e_1^{\Delta})$, $l \in \mathrm{supp}(e_2^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_2^{\Delta})$.
  * $l \notin \mathrm{supp}(e_1^{\mathrm{old}})$, $l \in \mathrm{supp}(e_1^{\Delta})$, $l \in \mathrm{supp}(e_2^{\mathrm{old}})$, $l \in \mathrm{supp}(e_2^{\Delta})$.
  * $l \in \mathrm{supp}(e_1^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_1^{\Delta})$, $l \notin \mathrm{supp}(e_2^{\mathrm{old}})$, $l \in \mathrm{supp}(e_2^{\Delta})$.
  * $l \in \mathrm{supp}(e_1^{\mathrm{old}})$, $l \in \mathrm{supp}(e_1^{\Delta})$, $l \notin \mathrm{supp}(e_2^{\mathrm{old}})$, $l \in \mathrm{supp}(e_2^{\Delta})$.
- Two cases follow from the fact that l only appears in $e_1^{\mathrm{old}}, e_2^{\mathrm{old}}$, or $e_1^{\Delta}, e_2^{\Delta}$, which are consistent.
  * $l \in \mathrm{supp}(e_1^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_1^{\Delta})$, $l \in \mathrm{supp}(e_2^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_2^{\Delta})$.
  * $l \notin \mathrm{supp}(e_1^{\mathrm{old}})$, $l \in \mathrm{supp}(e_1^{\Delta})$, $l \notin \mathrm{supp}(e_2^{\mathrm{old}})$, $l \in \mathrm{supp}(e_2^{\Delta})$.
- The final six cases follow immediately as l appears in dictionaries only on the left or only on the right hand side of label union.
  * $l \in \mathrm{supp}(e_1^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_1^{\Delta})$, $l \notin \mathrm{supp}(e_2^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_2^{\Delta})$.
  * $l \notin \mathrm{supp}(e_1^{\mathrm{old}})$, $l \in \mathrm{supp}(e_1^{\Delta})$, $l \notin \mathrm{supp}(e_2^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_2^{\Delta})$.
  * $l \in \mathrm{supp}(e_1^{\mathrm{old}})$, $l \in \mathrm{supp}(e_1^{\Delta})$, $l \notin \mathrm{supp}(e_2^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_2^{\Delta})$.
  * $l \notin \mathrm{supp}(e_1^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_1^{\Delta})$, $l \in \mathrm{supp}(e_2^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_2^{\Delta})$.
  * $l \notin \mathrm{supp}(e_1^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_1^{\Delta})$, $l \notin \mathrm{supp}(e_2^{\mathrm{old}})$, $l \in \mathrm{supp}(e_2^{\Delta})$.
  * $l \notin \mathrm{supp}(e_1^{\mathrm{old}})$, $l \notin \mathrm{supp}(e_1^{\Delta})$, $l \in \mathrm{supp}(e_2^{\mathrm{old}})$, $l \in \mathrm{supp}(e_2^{\Delta})$.

For the second part, relating the cost and degree of the delta to the cost and degree of the original query, the proofs are analogous to the cases from Theorem 2 and Theorem 4, when $h =$ **for** x **in** e_1 **union** e_2 and e_1 is input-independent, and $h = e_1 \uplus e_2$, respectively.

□

A.4 Delta-processing for simply-typed lambda calculi

A.4.1 Group structures over product and functional types

As it is essential for delta derivation that each type in the language has a commutative group structure, we use the operations of commutative groups over primitive types to inductively define similar structures for product and functional types as well:

$$\begin{array}{lll} \underline{0}_{A\times B} : 1 \to A\times B & \oplus_{A\times B} : (A\times B)^2 \to (A\times B) & \ominus_{A\times B} : (A\times B) \to (A\times B) \\ \underline{0}_{A\times B} = \langle \underline{0}_A, \underline{0}_B \rangle & \oplus_{A\times B} = (\oplus_A \times \oplus_B) \circ \mathrm{repair} & \ominus_{A\times B} = (\ominus_A \times \ominus_B) \end{array}$$

$$\begin{array}{lll} \underline{0}_{B^A} : 1 \to B^A & \oplus_{B^A} : (B^A)^2 \to B^A & \ominus_{B^A} : B^A \to B^A \\ \underline{0}_{B^A} = \mathrm{curry}(\underline{0}_B !_{1\times A}) & \oplus_{B^A} = \mathrm{curry}(\oplus_B \langle \mathrm{app} \circ (\pi_1 \times \mathrm{id}_A), & \ominus_{B^A} = \mathrm{curry}(\ominus_B \circ \mathrm{app}) \\ & \qquad \mathrm{app} \circ (\pi_2 \times \mathrm{id}_A) \rangle) & \end{array}$$

We extend addition over product values in a straightforward way by placing in each component of the output tuple the sum of the corresponding components from the input tuples. Similarly, the sum of two function values f_1, f_2, produces a function that returns for every possible input v the sum of $f_1(v)$ and $f_2(v)$. We show below that these definitions do indeed exhibit commutative group structures (lemmas 19 and 20). Additionally, the unit type can be seen as the commutative group containing just the neutral element: $(1, \underline{0}_1 = \mathrm{id}_1, \oplus_1 = !_{1\times 1}, \ominus_1 = \mathrm{id}_1)$.

We remark that function application distributes wrt. our definition of addition over functional values, thus facilitating its incrementalization:

$$\mathrm{app}(f \oplus_{B^A} df, a) = \mathrm{app}(f, a) \oplus_B \mathrm{app}(df, a)$$

Lemma 19. *If $(A, \underline{0}_A, \oplus_A, \ominus_A)$ and $(B, \underline{0}_B, \oplus_B, \ominus_B)$ are commutative groups, then $(A\times B, \underline{0}_{A\times B}, \oplus_{A\times B}, \ominus_{A\times B})$ is also a commutative group.*

Proof. This is a well known fact, but we include the proof for completeness.

- Associativity:

$$\begin{aligned}
&\oplus_{A\times B} \circ (\oplus_{A\times B} \times \mathrm{id}_{A\times B}) = \\
&= (\oplus_A \times \oplus_B) \circ \mathrm{repair} \circ (((\oplus_A \times \oplus_B) \circ \mathrm{repair}) \times \mathrm{id}_{A\times B}) \\
&= (\oplus_A \times \oplus_B) \circ \langle \langle \pi_{11}, \pi_{12} \rangle, \langle \pi_{21}, \pi_{22} \rangle \rangle \circ (((\oplus_A \times \oplus_B) \circ \langle \langle \pi_{11}, \pi_{12} \rangle, \langle \pi_{21}, \pi_{22} \rangle \rangle) \times \mathrm{id}_{A\times B}) \\
&= \langle \oplus_A \circ \langle \pi_{11}, \pi_{12} \rangle, \oplus_B \circ \langle \pi_{21}, \pi_{22} \rangle \rangle \circ (\langle \oplus_A \circ \langle \pi_{11}, \pi_{12} \rangle, \oplus_B \circ \langle \pi_{21}, \pi_{22} \rangle \rangle \times \mathrm{id}_{A\times B}) \\
&= \langle \oplus_A \circ \langle \pi_{11}, \pi_{12} \rangle, \oplus_B \circ \langle \pi_{21}, \pi_{22} \rangle \rangle \circ \langle \langle \oplus_A \circ \langle \pi_{111}, \pi_{121} \rangle, \oplus_B \circ \langle \pi_{211}, \pi_{221} \rangle \rangle, \pi_2 \rangle \\
&= \langle \oplus_A \circ \langle \oplus_A \circ \langle \pi_{111}, \pi_{121} \rangle, \pi_{12} \rangle, \oplus_B \circ \langle \oplus_B \circ \langle \pi_{211}, \pi_{221} \rangle, \pi_{22} \rangle \rangle
\end{aligned}$$

$$
\begin{aligned}
&= \langle \oplus_A \circ ((\oplus_A \circ (\pi_1 \times \pi_1)) \times \pi_1), \oplus_B \circ ((\oplus_B \circ (\pi_2 \times \pi_2)) \times \pi_2) \rangle \\
&= \langle \oplus_A \circ (\oplus_A \times \mathrm{id}_A) \circ ((\pi_1 \times \pi_1) \times \pi_1), \oplus_B \circ (\oplus_B \times \mathrm{id}_B) \circ ((\pi_2 \times \pi_2) \times \pi_2) \rangle \\
&= \langle \oplus_A \circ (\mathrm{id}_A \times \oplus_A) \circ \mathrm{rassoc}_\times \circ ((\pi_1 \times \pi_1) \times \pi_1), \\
&\quad\ \oplus_B \circ (\mathrm{id}_B \times \oplus_B) \circ \mathrm{rassoc}_\times \circ ((\pi_2 \times \pi_2) \times \pi_2) \rangle \\
&= \langle \oplus_A \circ (\mathrm{id}_A \times \oplus_A) \circ (\pi_1 \times (\pi_1 \times \pi_1)), \oplus_B \circ (\mathrm{id}_B \times \oplus_B) \circ (\pi_2 \times (\pi_2 \times \pi_2)) \rangle \circ \mathrm{rassoc}_\times \\
&= \langle \oplus_A \circ (\mathrm{id}_A \times \oplus_A) \circ \langle \pi_{11}, \langle \pi_{112}, \pi_{122} \rangle \rangle, \oplus_B \circ (\mathrm{id}_B \times \oplus_B) \circ \langle \pi_{21}, \langle \pi_{212}, \pi_{222} \rangle \rangle \rangle \circ \mathrm{rassoc}_\times \\
&= \langle \oplus_A \circ \langle \pi_{11}, \oplus_A \circ \langle \pi_{112}, \pi_{122} \rangle \rangle, \oplus_B \circ \langle \pi_{21}, \oplus_B \circ \langle \pi_{212}, \pi_{222} \rangle \rangle \rangle \circ \mathrm{rassoc}_\times \\
&= (\oplus_A \times \oplus_B) \circ \langle \langle \pi_{11}, \oplus_A \circ \langle \pi_{112}, \pi_{122} \rangle \rangle, \langle \pi_{21}, \oplus_B \circ \langle \pi_{212}, \pi_{222} \rangle \rangle \rangle \circ \mathrm{rassoc}_\times \\
&= (\oplus_A \times \oplus_B) \circ \mathrm{repair} \circ \langle \langle \pi_{11}, \pi_{21} \rangle, \langle \oplus_A \circ \langle \pi_{112}, \pi_{122} \rangle, \oplus_B \circ \langle \pi_{212}, \pi_{222} \rangle \rangle \rangle \circ \mathrm{rassoc}_\times \\
&= (\oplus_A \times \oplus_B) \circ \mathrm{repair} \circ (\mathrm{id}_{A\times B} \times \langle \oplus_A \circ \langle \pi_{11}, \pi_{12} \rangle, \oplus_B \circ \langle \pi_{21}, \pi_{22} \rangle \rangle) \circ \mathrm{rassoc}_\times \\
&= (\oplus_A \times \oplus_B) \circ \mathrm{repair} \circ (\mathrm{id}_{A\times B} \times ((\oplus_A \times \oplus_A) \circ \mathrm{repair})) \circ \mathrm{rassoc}_\times \\
&= \oplus_{A\times B} \circ (\mathrm{id}_{A\times B} \times \oplus_{A\times B}) \circ \mathrm{rassoc}_\times
\end{aligned}
$$

- Neutral Element:

$$
\begin{aligned}
&\oplus_{A\times B} \circ \langle \mathrm{id}_{A\times B}, \underline{0}_{A\times B} \circ !_{A\times B} \rangle \\
&= (\oplus_A \times \oplus_B) \circ \mathrm{repair} \circ \langle \mathrm{id}_A \times \mathrm{id}_B, (\underline{0}_A \circ !_A) \times (\underline{0}_B \circ !_B) \rangle \\
&= (\oplus_A \times \oplus_B) \circ (\langle \mathrm{id}_A, (\underline{0}_A \circ !_A) \rangle \times \langle \mathrm{id}_B, (\underline{0}_B \circ !_B) \rangle) \\
&= (\oplus_A \circ \langle \mathrm{id}_A, (\underline{0}_A \circ !_A) \rangle) \times (\oplus_B \circ \langle \mathrm{id}_B, (\underline{0}_B \circ !_B) \rangle) \\
&= \mathrm{id}_A \times \mathrm{id}_B = \mathrm{id}_{A\times B}
\end{aligned}
$$

  The proof of $\oplus_{A\times B} \circ \langle \underline{0}_{A\times B} \circ !_{A\times B}, \mathrm{id}_{A\times B} \rangle = \mathrm{id}_{A\times B}$ is analogous.

- Inverse element:

$$
\begin{aligned}
&\oplus_{A\times B} \circ \langle \mathrm{id}_{A\times B}, \ominus_{A\times B} \rangle \\
&= (\oplus_A \times \oplus_B) \circ \mathrm{repair} \circ \langle (\mathrm{id}_A \times \mathrm{id}_B), (\ominus_A \times \ominus_B) \rangle \\
&= (\oplus_A \times \oplus_B) \circ (\langle \mathrm{id}_A, \ominus_A \rangle \times \langle \mathrm{id}_B, \ominus_B \rangle) \\
&= (\oplus_A \circ \langle \mathrm{id}_A, \ominus_A \rangle) \times (\oplus_B \circ \langle \mathrm{id}_B, \ominus_B \rangle) \\
&= (\underline{0}_A \circ !_A) \times (\underline{0}_B \circ !_B) = \langle \underline{0}_A, \underline{0}_B \rangle \circ !_{A\times B} = \underline{0}_{A\times B} \circ !_{A\times B}
\end{aligned}
$$

  The proof of $\oplus_{A\times B} \circ \langle \ominus_{A\times B}, \mathrm{id}_{A\times B} \rangle = \underline{0}_{A\times B} \circ !_{A\times B}$ is analogous.

- Commutativity:

$$
\begin{aligned}
&\oplus_{A\times B} \circ \mathrm{sw}_\times \\
&= (\oplus_A \times \oplus_B) \circ \mathrm{repair} \circ \langle \pi_2, \pi_1 \rangle \\
&= (\oplus_A \times \oplus_B) \circ \langle \langle \pi_{11}, \pi_{12} \rangle, \langle \pi_{21}, \pi_{22} \rangle \rangle \circ \langle \pi_2, \pi_1 \rangle \\
&= (\oplus_A \times \oplus_B) \circ \langle \langle \pi_{12}, \pi_{11} \rangle, \langle \pi_{22}, \pi_{21} \rangle \rangle
\end{aligned}
$$

$$
\begin{aligned}
&= (\oplus_A \times \oplus_B) \circ (\mathrm{sw}_\times \times \mathrm{sw}_\times) \circ \langle \langle \pi_{11}, \pi_{12} \rangle, \langle \pi_{21}, \pi_{22} \rangle \rangle \\
&= ((\oplus_A \circ \mathrm{sw}_\times) \times (\oplus_B \circ \mathrm{sw}_\times)) \circ \mathrm{repair} \\
&= (\oplus_A \times \oplus_B) \circ \mathrm{repair} = \oplus_{A \times B}
\end{aligned}
$$

□

Lemma 20. *If $(B, \underline{0}_B, \oplus_B, \ominus_B)$ is a commutative group, then $(B^A, \underline{0}_{B^A}, \oplus_{B^A}, \ominus_{B^A})$ is also a commutative group.*

Proof. We use the fact that curry is a hom-set isomorphism, with $\mathrm{curry}^{-1}(g) = \mathrm{app} \circ (g \times \mathrm{id}_A)$, where $g : C \to B^A$. Therefore we can prove that $f_1 = f_2$ by proving that $\mathrm{curry}^{-1}(f_1) = \mathrm{curry}^{-1}(f_2)$.

- Associativity:

$$
\begin{aligned}
&\mathrm{curry}^{-1}(\oplus_{B^A} \circ (\oplus_{B^A} \times \mathrm{id}_{B^A})) = \\
&= \mathrm{app} \circ ((\oplus_{B^A} \circ (\oplus_{B^A} \times \mathrm{id}_{B^A})) \times \mathrm{id}_A) \\
&= \mathrm{app} \circ (\oplus_{B^A} \times \mathrm{id}_A) \circ ((\oplus_{B^A} \times \mathrm{id}_{B^A}) \times \mathrm{id}_A) \\
&= \oplus_B \circ \langle \mathrm{app} \circ (\pi_1 \times \mathrm{id}_A), \mathrm{app} \circ (\pi_2 \times \mathrm{id}_A) \rangle \circ ((\oplus_{B^A} \times \mathrm{id}_{B^A}) \times \mathrm{id}_A) \\
&= \oplus_B \circ \langle \mathrm{app} \circ ((\oplus_{B^A} \circ \pi_1) \times \mathrm{id}_A), \mathrm{app} \circ ((\mathrm{id}_{B^A} \circ \pi_2) \times \mathrm{id}_A) \rangle \\
&= \oplus_B \circ \langle \mathrm{app} \circ (\oplus_{B^A} \times \mathrm{id}_A) \circ (\pi_1 \times \mathrm{id}_A), \mathrm{app} \circ (\mathrm{id}_{B^A} \times \mathrm{id}_A) \circ (\pi_2 \times \mathrm{id}_A) \rangle \\
&= \oplus_B \circ \langle \oplus_B \circ \langle \mathrm{app} \circ (\pi_1 \times \mathrm{id}_A), \mathrm{app} \circ (\pi_2 \times \mathrm{id}_A) \rangle \circ (\pi_1 \times \mathrm{id}_A), \mathrm{app} \circ (\pi_2 \times \mathrm{id}_A) \rangle \\
&= \oplus_B \circ \langle \oplus_B \circ \langle \mathrm{app} \circ (\pi_{11} \times \mathrm{id}_A), \mathrm{app} \circ (\pi_{21} \times \mathrm{id}_A) \rangle, \mathrm{app} \circ (\pi_2 \times \mathrm{id}_A) \rangle \\
&= \oplus_B \circ (\oplus_B \times \mathrm{id}_B) \circ ((\mathrm{app} \times \mathrm{app}) \times \mathrm{app}) \circ \langle \langle \pi_{11} \times \mathrm{id}_A, \pi_{21} \times \mathrm{id}_A \rangle, \pi_2 \times \mathrm{id}_A \rangle \\
&= \oplus_B \circ (\mathrm{id}_B \times \oplus_B) \circ \mathrm{rassoc}_\times \circ ((\mathrm{app} \times \mathrm{app}) \times \mathrm{app}) \circ \langle \langle \pi_{11} \times \mathrm{id}_A, \pi_{21} \times \mathrm{id}_A \rangle, \pi_2 \times \mathrm{id}_A \rangle \\
&= \oplus_B \circ (\mathrm{id}_B \times \oplus_B) \circ (\mathrm{app} \times (\mathrm{app} \times \mathrm{app})) \circ \mathrm{rassoc}_\times \circ \langle \langle \pi_{11} \times \mathrm{id}_A, \pi_{21} \times \mathrm{id}_A \rangle, \pi_2 \times \mathrm{id}_A \rangle \\
&= \oplus_B \circ (\mathrm{id}_B \times \oplus_B) \circ (\mathrm{app} \times (\mathrm{app} \times \mathrm{app})) \circ \langle \pi_{11} \times \mathrm{id}_A, \langle \pi_{21} \times \mathrm{id}_A, \pi_2 \times \mathrm{id}_A \rangle \rangle \\
&= \oplus_B \circ (\mathrm{id}_B \times \oplus_B) \circ (\mathrm{app} \times (\mathrm{app} \times \mathrm{app})) \circ \langle \pi_1 \times \mathrm{id}_A, \langle \pi_{12} \times \mathrm{id}_A, \pi_{22} \times \mathrm{id}_A \rangle \rangle \circ \\
&\quad (\mathrm{rassoc}_\times \times \mathrm{id}_A) \\
&= \oplus_B \circ (\mathrm{app} \times (\oplus_B \circ (\mathrm{app} \times \mathrm{app}) \circ \langle \pi_1 \times \mathrm{id}_A, \pi_2 \times \mathrm{id}_A \rangle)) \circ \langle \pi_1 \times \mathrm{id}_A, \pi_2 \times \mathrm{id}_A \rangle \circ \\
&\quad (\mathrm{rassoc}_\times \times \mathrm{id}_A) \\
&= \oplus_B \circ (\mathrm{app} \times (\mathrm{app} \circ (\oplus_{B^A} \times \mathrm{id}_A))) \circ \langle \pi_1 \times \mathrm{id}_A, \pi_2 \times \mathrm{id}_A \rangle \circ (\mathrm{rassoc}_\times \times \mathrm{id}_A) \\
&= \oplus_B \circ (\mathrm{app} \times \mathrm{app}) \circ \langle \pi_1 \times \mathrm{id}_A, \pi_2 \times \mathrm{id}_A \rangle \circ ((\mathrm{id}_{B^A} \times \oplus_{B^A}) \times \mathrm{id}_A) \circ (\mathrm{rassoc}_\times \times \mathrm{id}_A) \\
&= \mathrm{app} \circ ((\oplus_{B^A} \circ (\mathrm{id}_{B^A} \times \oplus_{B^A}) \circ \mathrm{rassoc}_\times) \times \mathrm{id}_A) \\
&= \mathrm{curry}^{-1}(\oplus_{B^A} \circ (\mathrm{id}_{B^A} \times \oplus_{B^A}) \circ \mathrm{rassoc}_\times)
\end{aligned}
$$

- Neutral Element:

$$
\begin{aligned}
&\mathrm{curry}^{-1}(\oplus_{B^A} \circ \langle \mathrm{id}_{B^A}, \underline{0}_{B^A} \circ !_{B^A} \rangle) = \\
&= \mathrm{app} \circ ((\oplus_{B^A} \circ \langle \mathrm{id}_{B^A}, \underline{0}_{B^A} \circ !_{B^A} \rangle) \times \mathrm{id}_A) \\
&= \mathrm{app} \circ (\oplus_{B^A} \times \mathrm{id}_A) \circ (\langle \mathrm{id}_{B^A}, \underline{0}_{B^A} \circ !_{B^A} \rangle \times \mathrm{id}_A) \\
&= \oplus_B \circ (\mathrm{app} \times \mathrm{app}) \circ \langle \pi_1 \times \mathrm{id}_A, \pi_2 \times \mathrm{id}_A \rangle \circ (\langle \mathrm{id}_{B^A}, \underline{0}_{B^A} \circ !_{B^A} \rangle \times \mathrm{id}_A) \\
&= \oplus_B \circ (\mathrm{app} \times \mathrm{app}) \circ \langle \mathrm{id}_{B^A} \times \mathrm{id}_A, (\underline{0}_{B^A} \circ !_{B^A}) \times \mathrm{id}_A \rangle \\
&= \oplus_B \circ \langle \mathrm{app}, \mathrm{app} \circ (\underline{0}_{B^A} \times \mathrm{id}_A) \circ (!_{B^A} \times \mathrm{id}_A) \rangle \\
&= \oplus_B \circ \langle \mathrm{app}, \underline{0}_B \circ !_A \circ \pi_2 \circ (!_{B^A} \times \mathrm{id}_A) \rangle \\
&= \oplus_B \circ \langle \mathrm{app}, \underline{0}_B \circ !_A \circ \pi_2 \rangle = \oplus_B \circ \langle \mathrm{id}_B, \underline{0}_B \circ !_B \rangle \circ \mathrm{app} = \mathrm{app} = \mathrm{curry}^{-1}(\mathrm{id}_{B^A})
\end{aligned}
$$

The proof of $\oplus_{B^A} \circ \langle \underline{0}_{B^A} \circ !_{B^A}, \mathrm{id}_{B^A} \rangle = \mathrm{id}_{B^A}$ is analogous.

- Inverse Element:

$$
\begin{aligned}
&\mathrm{curry}^{-1}(\oplus_{B^A} \circ \langle \mathrm{id}_{B^A}, \ominus_{B^A} \rangle) = \\
&= \mathrm{app} \circ ((\oplus_{B^A} \circ \langle \mathrm{id}_{B^A}, \ominus_{B^A} \rangle) \times \mathrm{id}_A) \\
&= \mathrm{app} \circ (\oplus_{B^A} \times \mathrm{id}_A) \circ (\langle \mathrm{id}_{B^A}, \ominus_{B^A} \rangle \times \mathrm{id}_A) \\
&= \oplus_B \circ (\mathrm{app} \times \mathrm{app}) \circ \langle \pi_1 \times \mathrm{id}_A, \pi_2 \times \mathrm{id}_A \rangle \circ (\langle \mathrm{id}_{B^A}, \ominus_{B^A} \rangle \times \mathrm{id}_A) \\
&= \oplus_B \circ (\mathrm{app} \times \mathrm{app}) \circ \langle \mathrm{id}_{B^A} \times \mathrm{id}_A, \ominus_{B^A} \times \mathrm{id}_A \rangle \\
&= \oplus_B \circ \langle \mathrm{app}, \mathrm{app} \circ (\ominus_{B^A} \times \mathrm{id}_A) \rangle \\
&= \oplus_B \circ \langle \mathrm{app}, \ominus_B \circ \mathrm{app} \rangle = \oplus_B \circ \langle \mathrm{id}_B, \ominus_B \rangle \circ \mathrm{app} = \underline{0}_B \circ !_B \circ \mathrm{app} = \underline{0}_B \circ !_A \circ \pi_2 = \mathrm{curry}^{-1}(\underline{0}_{B^A})
\end{aligned}
$$

The proof of $\oplus_{B^A} \circ \langle \ominus_{B^A}, \mathrm{id}_{B^A} \rangle = \underline{0}_{B^A}$ is analogous.

- Commutativity:

$$
\begin{aligned}
&\mathrm{curry}^{-1}(\oplus_{B^A} \circ \mathrm{sw}_\times) = \\
&= \mathrm{app} \circ ((\oplus_{B^A} \circ \mathrm{sw}_\times) \times \mathrm{id}_A) \\
&= \mathrm{app} \circ (\oplus_{B^A} \times \mathrm{id}_A) \circ (\mathrm{sw}_\times \times \mathrm{id}_A) \\
&= \oplus_B \circ (\mathrm{app} \times \mathrm{app}) \circ \langle \pi_1 \times \mathrm{id}_A, \pi_2 \times \mathrm{id}_A \rangle \circ (\mathrm{sw}_\times \times \mathrm{id}_A) \\
&= \oplus_B \circ (\mathrm{app} \times \mathrm{app}) \circ \langle (\pi_1 \circ \mathrm{sw}_\times) \times \mathrm{id}_A, (\pi_2 \circ \mathrm{sw}_\times) \times \mathrm{id}_A \rangle \\
&= \oplus_B \circ (\mathrm{app} \times \mathrm{app}) \circ \langle \pi_2 \times \mathrm{id}_A, \pi_1 \times \mathrm{id}_A \rangle \\
&= \oplus_B \circ \mathrm{sw}_\times \circ (\mathrm{app} \times \mathrm{app}) \circ \langle \pi_1 \times \mathrm{id}_A, \pi_2 \times \mathrm{id}_A \rangle \\
&= \oplus_B \circ (\mathrm{app} \times \mathrm{app}) \circ \langle \pi_1 \times \mathrm{id}_A, \pi_2 \times \mathrm{id}_A \rangle = \mathrm{curry}^{-1}(\oplus_{B^A})
\end{aligned}
$$

$\square$

A.4.2 Deriving δ and cost functions

Theorem 12. *(Incrementality) For every $\mathcal{L}$ term $h : A \to B$*

$$h \circ \oplus_A = \oplus_B \langle h \circ \pi_1, \delta(h)\rangle,$$

given that this holds for every primitive in the language.

Proof. The proof follows by induction on the structure of h.

- $h = \mathrm{id}_A : \mathrm{id}_A \circ \oplus_A = \oplus_A \langle \pi_1, \pi_2\rangle = \oplus_A \langle \mathrm{id}_A \circ \pi_1, \delta(\mathrm{id}_A)\rangle$
- $h = g \circ f : (g \circ f) \circ \oplus_A = g \circ \oplus_B \langle f \circ \pi_1, \delta(f)\rangle = \oplus_C \langle g \circ \pi_1, \delta(g)\rangle \circ \langle f \circ \pi_1, \delta(f)\rangle = \oplus_C \langle (g \circ f) \circ \pi_1, \delta(g \circ f)\rangle$
- $h = !_A : !_A \circ \oplus_A = !_{A\times A} = !_{1\times 1} \circ \langle !_A \circ \pi_1, !_A \circ \pi_2\rangle = \oplus_1 \langle !_A \circ \pi_1, \delta(!_A)\rangle$
- $h = \langle f_1, f_2\rangle$

$$\begin{aligned}
&\langle f_1, f_2\rangle \circ \oplus_A = \langle f_1 \circ \oplus_A, f_2 \circ \oplus_A\rangle = \langle \oplus_{B_1}\langle f_1 \circ \pi_1, \delta(f_1)\rangle, \oplus_{B_2}\langle f_2 \circ \pi_1, \delta(f_2)\rangle\rangle = \\
&= (\oplus_{B_1} \times \oplus_{B_2}) \circ \langle\langle f_1 \circ \pi_1, \delta(f_1)\rangle, \langle f_2 \circ \pi_1, \delta(f_2)\rangle\rangle = \\
&= (\oplus_{B_1} \times \oplus_{B_2}) \circ \mathrm{repair} \circ \mathrm{repair} \circ \langle\langle f_1 \circ \pi_1, \delta(f_1)\rangle, \langle f_2 \circ \pi_1, \delta(f_2)\rangle\rangle = \\
&= \oplus_{B_1\times B_2}\langle\langle f_1 \circ \pi_1, f_2 \circ \pi_1\rangle, \langle \delta(f_1), \delta(f_2)\rangle\rangle = \oplus_{B_1\times B_2}\langle\langle f_1, f_2\rangle \circ \pi_1, \delta(\langle f_1, f_2\rangle)\rangle
\end{aligned}$$

- $h = \pi_i : \pi_i \circ \oplus_{B_1\times B_2} = \pi_i \circ (\oplus_{B_1} \times \oplus_{B_2}) \circ \mathrm{repair} = \oplus_{B_i} \circ \pi_i \circ \mathrm{repair} = \oplus_{B_i}\langle \pi_{i1}, \pi_{i2}\rangle = \oplus_{B_i}\langle \pi_i \circ \pi_1, \delta(\pi_i)\rangle$
- $h = \mathrm{curry}(f)$

  In order to prove this case we apply curry^{-1} on both sides and make use of the induction hypothesis on f.

$$\begin{aligned}
&\mathrm{curry}^{-1}(\mathrm{curry}(f) \circ \oplus_C) = \\
&= \mathrm{app} \circ ((\mathrm{curry}(f) \circ \oplus_C) \times \mathrm{id}_A) \\
&= \mathrm{app} \circ (\mathrm{curry}(f) \times \mathrm{id}_A) \circ (\oplus_C \times \mathrm{id}_A) \\
&= f \circ (\oplus_C \times \mathrm{id}_A) \\
&= f \circ (\oplus_C \times (\oplus_A \circ \langle \mathrm{id}_A, \underline{0}_A!\rangle)) \\
&= f \circ (\oplus_C \times \oplus_A) \circ (\mathrm{id}_{C\times C} \times \langle \mathrm{id}_A, \underline{0}_A!\rangle) \\
&= f \circ (\oplus_C \times \oplus_A) \circ \mathrm{repair} \circ \langle\langle \pi_{11}, \pi_2\rangle, \langle \pi_{21}, \underline{0}_A!\rangle\rangle \\
&= f \circ \oplus_{C\times A} \circ \langle\langle \pi_{11}, \pi_2\rangle, \langle \pi_{21}, \underline{0}_A!\rangle\rangle \\
&= \oplus_B\langle f \circ \pi_1, \delta(f)\rangle \circ \langle\langle \pi_{11}, \pi_2\rangle, \langle \pi_{21}, \underline{0}_A!\rangle\rangle \\
&= \oplus_B\langle f \circ \langle \pi_{11}, \pi_2\rangle, \delta(f) \circ \langle\langle \pi_{11}, \pi_2\rangle, \langle \pi_{21}, \underline{0}_A!\rangle\rangle\rangle \\
&= \oplus_B\langle \mathrm{app} \circ (\mathrm{curry}(f) \times \mathrm{id}_A) \circ (\pi_1 \times \mathrm{id}_A), \mathrm{app} \circ (\delta(\mathrm{curry}(f)) \times \mathrm{id}_A)\rangle
\end{aligned}$$

$$
\begin{aligned}
&= \oplus_B\langle \mathrm{app}\circ(\pi_1 \times \mathrm{id}_A), \mathrm{app}\circ(\pi_2 \times \mathrm{id}_A)\rangle \circ (\langle \mathrm{curry}(f)\circ\pi_1, \delta(\mathrm{curry}(f))\rangle \times \mathrm{id}_A)\\
&= \mathrm{app}\circ(\oplus_{B^A} \times \mathrm{id}_A)\circ(\langle \mathrm{curry}(f)\circ\pi_1, \delta(\mathrm{curry}(f))\rangle \times \mathrm{id}_A)\\
&= \mathrm{app}\circ(\oplus_{B^A}\langle \mathrm{curry}(f)\circ\pi_1, \delta(\mathrm{curry}(f))\rangle \times \mathrm{id}_A)\\
&= \mathrm{curry}^{-1}(\oplus_{B^A}\langle \mathrm{curry}(f)\circ\pi_1, \delta(\mathrm{curry}(f))\rangle)
\end{aligned}
$$

- $h = \mathrm{app}$

  In order to prove this case we show that for any $f : C \times A \to B$, s.t.: $f \circ \oplus_{C\times A} = \oplus_B\langle f\circ\pi_1, \delta(f)\rangle$, the following holds:

$$
\mathrm{app}\circ\oplus_{B^A\times A}\circ((\mathrm{curry}(f)\times\mathrm{id}_A)\times\mathrm{id}_{B^A\times A}) = \oplus_B\circ\langle \mathrm{app}\circ\pi_1, \delta(\mathrm{app})\rangle\circ((\mathrm{curry}(f)\times\mathrm{id}_A)\times\mathrm{id}_{B^A\times A}).
$$

Left-side:

$$
\begin{aligned}
&\mathrm{app}\circ\oplus_{B^A\times A}((\mathrm{curry}(f)\times\mathrm{id}_A)\times\mathrm{id}_{B^A\times A}) =\\
&= \mathrm{app}\circ(\oplus_{B^A}\times\oplus_A)\circ\mathrm{repair}\circ((\mathrm{curry}(f)\times\mathrm{id}_A)\times\mathrm{id}_{B^A\times A})\\
&= \mathrm{app}\circ(\oplus_{B^A}\times\mathrm{id}_A)\circ(\mathrm{id}_{B^A\times B^A}\times\oplus_A)\circ((\mathrm{curry}(f)\times\mathrm{id}_{B^A})\times\mathrm{id}_{A\times A})\circ\mathrm{repair}\\
&= \oplus_B\langle \mathrm{app}\circ(\pi_1\times\mathrm{id}_A), \mathrm{app}\circ(\pi_2\times\mathrm{id}_A)\rangle\circ((\mathrm{curry}(f)\times\mathrm{id}_{B^A})\times\oplus_A)\circ\mathrm{repair}\\
&= \oplus_B\langle \mathrm{app}\circ(\mathrm{curry}(f)\times\mathrm{id}_A)\circ(\pi_1\times\oplus_A), \mathrm{app}\circ(\pi_2\times\oplus_A)\rangle\circ\mathrm{repair}\\
&= \oplus_B\langle f\circ(\pi_1\times\oplus_A), \mathrm{app}\circ(\pi_2\times\oplus_A)\rangle\circ\mathrm{repair}\\
&= \oplus_B\langle f\circ(\oplus_C\times\oplus_A)\circ(\langle\pi_1, \underline{0}_C!\rangle\times\mathrm{id}_{A\times A}), \mathrm{app}\circ(\pi_2\times\oplus_A)\rangle\circ\mathrm{repair}\\
&= \oplus_B\langle f\circ\oplus_{C\times A}\circ\langle\langle\pi_{11}, \pi_{12}\rangle, \langle\underline{0}_C!, \pi_{22}\rangle\rangle, \mathrm{app}\circ(\pi_2\times\oplus_A)\rangle\circ\mathrm{repair}\\
&= \oplus_B\ \langle\oplus_B\langle f\circ\pi_1, \delta(f)\rangle\circ\langle\langle\pi_{11}, \pi_{12}\rangle, \langle\underline{0}_C!, \pi_{22}\rangle\rangle, \mathrm{app}\circ(\pi_2\times\oplus_A)\rangle\circ\mathrm{repair}\\
&= \oplus_B\langle\oplus_B\langle f\circ\langle\pi_{11}, \pi_{12}\rangle, \delta(f)\circ\langle\langle\pi_{11}, \pi_{12}\rangle, \langle\underline{0}_C!, \pi_{22}\rangle\rangle\rangle, \mathrm{app}\circ(\pi_2\times\oplus_A)\rangle\circ\mathrm{repair}\\
&= \oplus_B\langle f\circ\langle\pi_{11}, \pi_{12}\rangle, \oplus_B\langle\delta(f)\circ\langle\langle\pi_{11}, \pi_{12}\rangle, \langle\underline{0}_C!, \pi_{22}\rangle\rangle, \mathrm{app}\circ(\pi_2\times\oplus_A)\rangle\rangle\circ\mathrm{repair}\\
&= \oplus_B\langle f\circ\pi_1, \oplus_B\langle\delta(f)\circ\langle\langle\pi_{11}, \pi_{12}\rangle, \langle\underline{0}_C!, \pi_{22}\rangle\rangle, \mathrm{app}\circ(\pi_2\times\oplus_A)\rangle\circ\mathrm{repair}\rangle
\end{aligned}
$$

Right-side:

$$
\begin{aligned}
&\oplus_B\ \langle\mathrm{app}\circ\pi_1, \delta(\mathrm{app})\rangle\circ((\mathrm{curry}(f)\times\mathrm{id}_A)\times\mathrm{id}_{B^A\times A}) =\\
&= \oplus_B\langle\mathrm{app}\circ(\mathrm{curry}(f)\times\mathrm{id}_A)\circ\pi_1, \delta(\mathrm{app})\circ((\mathrm{curry}(f)\times\mathrm{id}_A)\times\mathrm{id}_{B^A\times A})\rangle\\
&= \oplus_B\langle f\circ\pi_1, \oplus_B\langle\mathrm{app}\circ((\mathrm{delta}_0\circ\pi_1)\times\mathrm{id}_{A\times A}), \mathrm{app}\circ(\pi_2\times\oplus_A)\rangle\circ\\
&\qquad\qquad \mathrm{repair}\circ((\mathrm{curry}(f)\times\mathrm{id}_A)\times\mathrm{id}_{B^A\times A})\rangle\\
&= \oplus_B\langle f\circ\pi_1, \oplus_B\langle\mathrm{app}\circ((\mathrm{delta}_0\circ\pi_1)\times\mathrm{id}_{A\times A}), \mathrm{app}\circ(\pi_2\times\oplus_A)\rangle\circ\\
&\qquad\qquad ((\mathrm{curry}(f)\times\mathrm{id}_{B^A})\times\mathrm{id}_{A\times A})\circ\mathrm{repair}\rangle\\
&= \oplus_B\langle f\circ\pi_1, \oplus_B\langle\mathrm{app}\circ((\mathrm{delta}_0\circ\mathrm{curry}(f)\circ\pi_1)\times\mathrm{id}_{A\times A}), \mathrm{app}\circ(\pi_2\times\oplus_A)\rangle\circ\mathrm{repair}\rangle\\
&= \oplus_B\langle f\circ\pi_1, \oplus_B\langle\mathrm{app}\circ((\mathrm{delta}_0\circ\mathrm{curry}(f))\times\mathrm{id}_{A\times A})\circ(\pi_1\times\mathrm{id}_{A\times A}), \mathrm{app}\circ(\pi_2\times\oplus_A)\rangle\circ\\
&\qquad\qquad \mathrm{repair}\rangle\\
&= \oplus_B\langle f\circ\pi_1, \oplus_B\langle\delta(f)\circ\langle\langle\pi_1, \pi_{12}\rangle, \langle\underline{0}_C!, \pi_{22}\rangle\rangle\circ(\pi_1\times\mathrm{id}_{A\times A}), \mathrm{app}\circ(\pi_2\times\oplus_A)\rangle\circ\mathrm{repair}\rangle
\end{aligned}
$$

$$= \oplus_B \langle f \circ \pi_1, \oplus_B \langle \delta(f) \circ \langle \langle \pi_{11}, \pi_{12} \rangle, \langle \underline{0}_C !, \pi_{22} \rangle \rangle, \mathrm{app} \circ (\pi_2 \times \oplus_A) \rangle \circ \mathrm{repair} \rangle$$

- $h = \mathrm{delta_o}$

$$\begin{aligned}
&\mathrm{delta_o} \circ \oplus_{B^A} \langle \mathrm{curry}(f), \mathrm{curry}(df) \rangle = \mathrm{delta_o} \circ \mathrm{curry}(\oplus_B \langle f, df \rangle) = \\
&= \mathrm{curry}(\delta_{-,A}(\oplus_B \langle f, df \rangle)) = \mathrm{curry}(\oplus_B \circ \delta_{-,A}(\langle f, df \rangle)) = \\
&= \mathrm{curry}(\oplus_B \langle \delta_{-,A}(f), \delta_{-,A}(df) \rangle) = \oplus_{B^{A \times A}} \langle \mathrm{curry}(\delta_{-,A}(f)), \mathrm{curry}(\delta_{-,A}(df)) \rangle = \\
&= \oplus_{B^{A \times A}} \langle \mathrm{delta_o} \circ \mathrm{curry}(f), \mathrm{delta_o} \circ \mathrm{curry}(df) \rangle \\
&= \oplus_{B^{A \times A}} \langle \mathrm{delta_o} \circ \pi_1, \mathrm{delta_o} \circ \pi_2 \rangle \circ \langle \mathrm{curry}(f), \mathrm{curry}(df) \rangle \\
&= \oplus_{B^{A \times A}} \langle \mathrm{delta_o} \circ \pi_1, \delta(\mathrm{delta_o}) \rangle \circ \langle \mathrm{curry}(f), \mathrm{curry}(df) \rangle
\end{aligned}$$

- $h = \underline{0}_{\mathbb{D}}$

$$\begin{aligned}
&\underline{0}_{\mathbb{D}} \circ \oplus_1 = \underline{0}_{\mathbb{D}} \circ !_{1 \times 1} = \underline{0}_{\mathbb{D}} \circ \pi_1 = \mathrm{id}_{\mathbb{D}} \circ \underline{0}_{\mathbb{D}} \circ \pi_1 = \oplus_{\mathbb{D}} \langle \mathrm{id}_{\mathbb{D}}, \underline{0}_{\mathbb{D}} !_{\mathbb{D}} \rangle \circ \underline{0}_{\mathbb{D}} \circ \pi_1 = \oplus_{\mathbb{D}} \langle \underline{0}_{\mathbb{D}} \circ \pi_1, \underline{0}_{\mathbb{D}} !_{1 \times 1} \rangle = \\
&= \oplus_{\mathbb{D}} \langle \underline{0}_{\mathbb{D}} \circ \pi_1, \underline{0}_{\mathbb{D}} \circ \pi_2 \rangle = \oplus_{\mathbb{D}} \langle \underline{0}_{\mathbb{D}} \circ \pi_1, \delta(\underline{0}_{\mathbb{D}}) \rangle
\end{aligned}$$

- $h = \oplus_{\mathbb{D}}$

$$\begin{aligned}
&\oplus_{\mathbb{D}} \circ \oplus_{\mathbb{D} \times \mathbb{D}} = \oplus_{\mathbb{D}} \circ (\oplus_{\mathbb{D}} \times \oplus_{\mathbb{D}}) \circ \mathrm{repair} = \oplus_{\mathbb{D}} \langle \oplus_{\mathbb{D}} \langle \pi_{11}, \pi_{12} \rangle, \oplus_{\mathbb{D}} \langle \pi_{21}, \pi_{22} \rangle \rangle = \\
&= \oplus_{\mathbb{D}} \langle \oplus_{\mathbb{D}} \langle \pi_{11}, \pi_{21} \rangle, \oplus_{\mathbb{D}} \langle \pi_{12}, \pi_{22} \rangle \rangle = \oplus_{\mathbb{D}} \langle \oplus_{\mathbb{D}} \circ \pi_1, \oplus_{\mathbb{D}} \circ \pi_2 \rangle = \oplus_{\mathbb{D}} \langle \oplus_{\mathbb{D}} \circ \pi_1, \delta(\oplus_{\mathbb{D}}) \rangle
\end{aligned}$$

- $h = \ominus_{\mathbb{D}}$

  We prove this case for arbitrary $a, b \in \mathbb{D}$ and we use $\oplus_{\mathbb{D}}$ in infix form.

$$\begin{aligned}
&\ominus_{\mathbb{D}} (a \oplus_{\mathbb{D}} b) = (\ominus_{\mathbb{D}} (a \oplus_{\mathbb{D}} b)) \oplus_{\mathbb{D}} \underline{0}_{\mathbb{D}} \oplus_{\mathbb{D}} \underline{0}_{\mathbb{D}} = \\
&= (\ominus_{\mathbb{D}} (a \oplus_{\mathbb{D}} b)) \oplus_{\mathbb{D}} (a \oplus_{\mathbb{D}} (\ominus_{\mathbb{D}} a)) \oplus_{\mathbb{D}} (b \oplus_{\mathbb{D}} (\ominus_{\mathbb{D}} b)) \\
&= (\ominus_{\mathbb{D}} (a \oplus_{\mathbb{D}} b)) \oplus_{\mathbb{D}} (a \oplus_{\mathbb{D}} b) \oplus_{\mathbb{D}} ((\ominus_{\mathbb{D}} a) \oplus_{\mathbb{D}} (\ominus_{\mathbb{D}} b)) \\
&= \underline{0}_{\mathbb{D}} \oplus_{\mathbb{D}} ((\ominus_{\mathbb{D}} a) \oplus_{\mathbb{D}} (\ominus_{\mathbb{D}} b)) = ((\ominus_{\mathbb{D}} a) \oplus_{\mathbb{D}} (\ominus_{\mathbb{D}} b))
\end{aligned}$$

□

Theorem 13. *If every primitive* udef *is efficiently incrementalizable, then the same property holds for the entire language* $\mathcal{L}(\mathbb{D}, \underline{\mathrm{udef}})$*, where a input-dependent term* $h : A \to B$ *is* efficiently incrementalizable *if* $\forall \epsilon, \epsilon_\Delta \in A^\circ$ *s.t.* $\epsilon_\Delta \prec \epsilon$*, then*

$$\mathrm{cost}(\delta(h))(\epsilon, \epsilon_\Delta) \prec \mathrm{cost}(h)(\epsilon).$$

Proof. The proof follows by induction on the structure of h.

- $h = \mathrm{id}_A$

$$\mathrm{cost}(\delta(\mathrm{id}_A))(\epsilon_A,\epsilon_{\Delta A}) = \mathrm{cost}(\pi_2)(\epsilon_A,\epsilon_{\Delta A}) = \pi_2(\epsilon_A,\epsilon_{\Delta A}) = \epsilon_{\Delta A} \preceq_A \epsilon_A = \mathrm{cost}(\mathrm{id}_A)(\epsilon_A)$$

- $h = g \circ f$

$$\begin{aligned}
&\mathrm{cost}(\delta(g\circ f))(\epsilon_A,\epsilon_{\Delta A}) = \mathrm{cost}(\delta(g)\circ\langle f\circ\pi_1,\delta(f)\rangle))(\epsilon_A,\epsilon_{\Delta A}) = \\
&= \mathrm{cost}(\delta(g))\circ\langle \mathrm{cost}(f)\circ\pi_1,\mathrm{cost}(\delta(f))\rangle))(\epsilon_A,\epsilon_{\Delta A}) = \\
&= \mathrm{cost}(\delta(g))(\mathrm{cost}(f)(\epsilon_A),\mathrm{cost}(\delta(f))(\epsilon_A,\epsilon_{\Delta A})) \\
&\preceq_C \mathrm{cost}(g)(\mathrm{cost}(f)(\epsilon_A)) = \mathrm{cost}(g\circ f)(\epsilon_A)
\end{aligned}$$

- $h = !_A$

$$\begin{aligned}
&\mathrm{cost}(\delta(!_A))(\epsilon_A,\epsilon_{\Delta A}) = \mathrm{cost}(!_A\circ\pi_2)(\epsilon_A,\epsilon_{\Delta A}) = (\mathrm{cost}(!_A)\circ\pi_2)(\epsilon_A,\epsilon_{\Delta A}) = \\
&= \mathrm{cost}(!_A)(\epsilon_{\Delta A}) \preceq_1 \mathrm{cost}(!_A)(\epsilon_A)
\end{aligned}$$

- $h = \langle f_1, f_2\rangle$

$$\begin{aligned}
&\mathrm{cost}(\delta(\langle f_1,f_2\rangle))(\epsilon_A,\epsilon_{\Delta A}) = \mathrm{cost}(\langle\delta(f_1),\delta(f_2)\rangle)(\epsilon_A,\epsilon_{\Delta A}) = \\
&= \langle\mathrm{cost}(\delta(f_1)),\mathrm{cost}(\delta(f_2))\rangle(\epsilon_A,\epsilon_{\Delta A}) = \langle\mathrm{cost}(\delta(f_1))(\epsilon_A,\epsilon_{\Delta A}),\mathrm{cost}(\delta(f_2))(\epsilon_A,\epsilon_{\Delta A})\rangle \\
&\preceq_{B_1\times B_2} \langle\mathrm{cost}(f_1)(\epsilon_A),\mathrm{cost}(f_2)(\epsilon_A)\rangle = \langle\mathrm{cost}(f_1),\mathrm{cost}(f_2)\rangle(\epsilon_A) = \mathrm{cost}(\langle f_1,f_2\rangle)(\epsilon_A)
\end{aligned}$$

- $h = \pi_i$

$$\begin{aligned}
&\mathrm{cost}(\delta(\pi_i))(\langle\epsilon_{B_1},\epsilon_{B_2}\rangle,\langle\epsilon_{\Delta B_1},\epsilon_{\Delta B_2}\rangle) = \mathrm{cost}(\pi_i\circ\pi_2)(\langle\epsilon_{B_1},\epsilon_{B_2}\rangle,\langle\epsilon_{\Delta B_1},\epsilon_{\Delta B_2}\rangle) = \\
&= \epsilon_{\Delta B_i} \preceq_{B_i} \epsilon_{B_i} = \mathrm{cost}(\pi_i)(\langle\epsilon_{B_1},\epsilon_{B_2}\rangle)
\end{aligned}$$

- $h = \mathrm{curry}(f)$. We show that for any $\epsilon_A \in A^\circ$ such that:
  $\mathrm{cost}(\delta(\mathrm{curry}(f)))(\epsilon_C,\epsilon_{\Delta C})(\epsilon_A) \preceq_B \mathrm{cost}(\mathrm{curry}(f))(\epsilon_C)(\epsilon_A)$, then:

$$\begin{aligned}
&\mathrm{cost}(\delta(\mathrm{curry}(f)))(\epsilon_C,\epsilon_{\Delta C})(\epsilon_A) = \\
&= \mathrm{cost}(\mathrm{curry}(\delta(f)\circ\langle\langle\pi_{11},\pi_2\rangle,\langle\pi_{21},\underline{0}_A!\rangle\rangle))(\epsilon_C,\epsilon_{\Delta C})(\epsilon_A) = \\
&= \mathrm{curry}(\mathrm{cost}(\delta(f))\circ\langle\langle\pi_{11},\pi_2\rangle,\langle\pi_{21},\underline{1}_A!\rangle\rangle)(\epsilon_C,\epsilon_{\Delta C})(\epsilon_A) = \\
&= \mathrm{cost}(\delta(f))(\langle\epsilon_C,\epsilon_A\rangle,\langle\epsilon_{\Delta C},\underline{1}_A!\rangle) \\
&\preceq_B \mathrm{cost}(f)(\langle\epsilon_C,\epsilon_A\rangle) = \mathrm{curry}(\mathrm{cost}(f))(\epsilon_C)(\epsilon_A) = \mathrm{cost}(\mathrm{curry}(f))(\epsilon_C)(\epsilon_A)
\end{aligned}$$

- $h = \mathrm{app}$

  Left side:

$$\mathrm{cost}(\delta(\mathrm{app}))(\langle\epsilon_{B^A},\epsilon_A\rangle,\langle\epsilon_{\Delta B^A},\epsilon_{\Delta A}\rangle) =$$

$$
\begin{aligned}
&= \mathrm{cost}(\oplus_B \circ \langle \mathrm{app} \circ ((\mathrm{delta_o} \circ \pi_1) \times \mathrm{id}_{A \times A}), \mathrm{app} \circ (\pi_2 \times \oplus_A) \rangle \circ \mathrm{repair}) \\
&\quad (\langle \epsilon_{B^A}, \epsilon_A \rangle, \langle \epsilon_{\Delta B^A}, \epsilon_{\Delta A} \rangle) \\
&= (\max_B \circ \langle \mathrm{app} \circ ((\mathrm{cost}(\mathrm{delta_o}) \circ \pi_1) \times \mathrm{id}_{A^\circ \times A^\circ}), \mathrm{app} \circ (\pi_2 \times \max_A) \rangle \circ \mathrm{repair}) \\
&\quad (\langle \epsilon_{B^A}, \epsilon_A \rangle, \langle \epsilon_{\Delta B^A}, \epsilon_{\Delta A} \rangle) \\
&= \max_B(\mathrm{cost}(\mathrm{delta_o})(\epsilon_{B^A})(\langle \epsilon_A, \epsilon_{\Delta A} \rangle), \epsilon_{\Delta B^A}(\epsilon_A))
\end{aligned}
$$

Right side:

$$\mathrm{cost}(\mathrm{app})(\langle \epsilon_{B^A}, \epsilon_A \rangle) = \mathrm{app}(\langle \epsilon_{B^A}, \epsilon_A \rangle) = \epsilon_{B^A}(\epsilon_A)$$

The result follows from $\epsilon_{\Delta B^A} \preceq_{B^A} \epsilon_{B^A}$ and the induction hypothesis on $f : C \times A \to B$ s.t. $\epsilon_{B^A} = \mathrm{curry}(\mathrm{cost}(f))(\epsilon_C)$:

$$
\begin{aligned}
&\mathrm{cost}(\mathrm{delta_o})(\epsilon_{B^A})(\langle \epsilon_A, \epsilon_{\Delta A} \rangle) = \\
&= \mathrm{cost}(\mathrm{delta_o})(\mathrm{curry}(\mathrm{cost}(f))(\epsilon_C))(\langle \epsilon_A, \epsilon_{\Delta A} \rangle) \\
&= \mathrm{cost}(\mathrm{delta_o} \circ \mathrm{curry}(f))(\epsilon_C)(\langle \epsilon_A, \epsilon_{\Delta A} \rangle) \\
&= \mathrm{cost}(\mathrm{curry}(\delta(f) \circ \langle \langle \pi_1, \pi_{12} \rangle, \langle \underline{0}_C \circ !, \pi_{22} \rangle \rangle))(\epsilon_C)(\langle \epsilon_A, \epsilon_{\Delta A} \rangle) \\
&= \mathrm{curry}(\mathrm{cost}(\delta(f)) \circ \langle \langle \pi_1, \pi_{12} \rangle, \langle \underline{1}_C \circ !, \pi_{22} \rangle \rangle)(\epsilon_C)(\langle \epsilon_A, \epsilon_{\Delta A} \rangle) \\
&= \mathrm{cost}(\delta(f))(\langle \epsilon_C, \epsilon_A \rangle, \langle \underline{1}_C \circ !, \epsilon_{\Delta A} \rangle) \\
&\preceq_B \mathrm{cost}(f)(\langle \epsilon_C, \epsilon_A \rangle) = \mathrm{curry}(\mathrm{cost}(f))(\epsilon_C)(\epsilon_A) = \epsilon_{B^A}(\epsilon_A)
\end{aligned}
$$

- $h = \underline{0}_{\mathbb{D}}$

$$\mathrm{cost}(\delta(\underline{0}_{\mathbb{D}}))(\epsilon_1, \epsilon_{\Delta 1}) = \mathrm{cost}(\underline{0}_{\mathbb{D}} \circ \pi_2)(\epsilon_1, \epsilon_{\Delta 1}) = \underline{1}_{\mathbb{D}}(\epsilon_{\Delta 1}) \preceq_{\mathbb{D}} \underline{1}_{\mathbb{D}}(\epsilon_1) = \mathrm{cost}(\underline{0}_{\mathbb{D}})(\epsilon_1)$$

- $h = \oplus_{\mathbb{D}}$

$$
\begin{aligned}
&\mathrm{cost}(\delta(\oplus_{\mathbb{D}}))(\langle \epsilon_{1\mathbb{D}}, \epsilon_{2\mathbb{D}} \rangle, \langle \epsilon_{1\Delta\mathbb{D}}, \epsilon_{2\Delta\mathbb{D}} \rangle) = \mathrm{cost}(\oplus_{\mathbb{D}} \circ \pi_2)(\langle \epsilon_{1\mathbb{D}}, \epsilon_{2\mathbb{D}} \rangle, \langle \epsilon_{1\Delta\mathbb{D}}, \epsilon_{2\Delta\mathbb{D}} \rangle) = \\
&= \mathrm{cost}(\oplus_{\mathbb{D}})(\langle \epsilon_{1\Delta\mathbb{D}}, \epsilon_{2\Delta\mathbb{D}} \rangle) = \max_{\mathbb{D}}(\langle \epsilon_{1\Delta\mathbb{D}}, \epsilon_{2\Delta\mathbb{D}} \rangle) \\
&\preceq_{\mathbb{D}} \max_{\mathbb{D}}(\langle \epsilon_{1\mathbb{D}}, \epsilon_{2\mathbb{D}} \rangle) = \mathrm{cost}(\oplus_{\mathbb{D}})(\langle \epsilon_{1\mathbb{D}}, \epsilon_{2\mathbb{D}} \rangle)
\end{aligned}
$$

- $h = \ominus_{\mathbb{D}}$

$$
\begin{aligned}
&\mathrm{cost}(\delta(\ominus_{\mathbb{D}}))(\epsilon_{\mathbb{D}}, \epsilon_{\Delta\mathbb{D}}) = \mathrm{cost}(\ominus_{\mathbb{D}} \circ \pi_2)(\epsilon_{\mathbb{D}}, \epsilon_{\Delta\mathbb{D}}) = \mathrm{cost}(\ominus_{\mathbb{D}})(\epsilon_{\Delta\mathbb{D}}) = \mathrm{id}_{\mathbb{N}^+}(\epsilon_{\Delta\mathbb{D}}) = \epsilon_{\Delta\mathbb{D}} \\
&\preceq_{\mathbb{D}} \epsilon_{\mathbb{D}} = \mathrm{id}_{\mathbb{N}^+}(\epsilon_{\mathbb{D}}) = \mathrm{cost}(\ominus_{\mathbb{D}})(\epsilon_{\mathbb{D}})
\end{aligned}
$$

□

A.5 SLeNDer Benchmark

Below we present the definitions of the queries used in our benchmark. We provide a version based strictly on comprehension syntax (as we consider it more readable), as well as a Spark version that works around Spark's restriction wrt. referencing RDDs within the body of a comprehension.

```
// Q1 - comprehension query
var Q1 =
  for (c <- Customer)
  yield (c.name,
      for (o <- Orders
           if o.custkey == c.custkey)
      yield (o.orderdate,
          for (l <- Lineitem; p <- Part
               if l.orderkey == o.orderkey &&
                  l.partkey == p.partkey)
          yield p.name))

// Q1 - Spark query
var OrderParts = (
  for ((_, (l_orderkey, p_name)) <-
         Lineitem.map(l => l.partkey -> l.orderkey)
             .join( Part.map(p => p.partkey -> p.name) ) )
  yield (l_orderkey, p_name) )
  .groupByKey()

var CustomerOrders = (
  for ((_, ((o_custkey, o_orderdate), parts)) <-
        Orders.map(o => o.orderkey->(o.custkey, o.orderdate))
              .join(OrderParts) )
  yield (o.custkey, (o_orderdate, parts)) )
  .groupByKey()

var Q1 =
  for ((_, (c_name,orders)) <-
         Customer.map(c => c.custkey -> c.name)
                 .join(CustomerOrders) )
  yield (c_name, orders)

////////////////////////////////////////////////////////////
// Q2 - comprehension query
var Q2 =
  for (s <- Supplier)
  yield(s.name,
       for (l <- Lineitem if s.suppkey == l.suppkey;
            o <- Orders   if o.orderkey == l.orderkey;
            c <- Customer if c.custkey == o.custkey)
       yield c.name)

// Q2 - spark query
var OrderLineitems =
  for ((_, (o_custkey,l_suppkey)) <-
         Orders.map(o => o.orderkey -> o.custkey)
           .join( Lineitem.map(l => l.orderkey->l.suppkey) ))
  yield (o_custkey,l_suppkey)

var SupplierCustomers = (
  for ((_, (c_name,l_suppkey)) <-
           Customer.map(c => c.custkey -> c.name)
               .join( OrderLineitems ))
  yield (l_suppkey,c_name) )
  .groupByKey()

var Q2 =
  for ((_, (s_name, customers)) <-
            Supplier.map(s => s.suppkey -> s.name)
                    .join( SupplierCustomers ))
  yield (s_name, customers)

////////////////////////////////////////////////////////////
// Q3 - comprehension query
var Q3 =
  for (p <- Part)
  yield (p.name,
      for (ps <- PartSupp if ps.partkey == p.partkey;
            s <- Supplier if s.suppkey == ps.suppkey)
      yield s.name,
      for (l <- Lineitem if l.partkey == p.partkey;
           o <- Orders if o.orderkey == l.orderkey;
           c <- Customer if c.custkey == o.custkey)
      yield c.name)

// Q3 - Spark query
var PartSuppliers = (
  for (ps <- PartSupp;
        s <- Supplier if s.suppkey == ps.suppkey)
  yield (ps.partkey, s.name) )
  .groupByKey()

var PartCustomers = (
  for (l <- Lineitem;
       o <- Orders if o.orderkey == l.orderkey;
       c <- Customer if c.custkey == o.custkey)
  yield (l.partkey, c.name) )
  .groupByKey()

var Q3 =
  for ((_, (p_name, (suppliers,customers))) <-
          Part.map(p => p.partkey -> p.name)
              .join( PartSuppliers.join(PartCustomers) ))
  yield (p_name, suppliers, customers)

////////////////////////////////////////////////////////////
// Q4 - comprehension/Spark query
var Q4 = (
  for ((c_name, c_orders) <- Q1;
       (o_orderdate, o_parts) <- c_orders;
       (p_name, l_qty) <- o_parts)
  yield ((c_name,p_name,getMonth(o_orderdate)),l_qty) )
  .reduceByKey(_ + _)

////////////////////////////////////////////////////////////
// Q5 - comprehension/Spark query
var Q5 = (
  for ((p_name, suppliers, customers) <- Q3;
       (c_name,c_nationkey) <- customers
       if suppliers.forall{ case (_,s_nationkey) =>
             c_nationkey != s_nationkey })
  yield (p_name, 1) )
  .reduceByKey(_ + _)

////////////////////////////////////////////////////////////
// Q6 - comprehension query
var Q6 =
  for ((c_name,_) <- Customer)
  yield (c_name,
      for ((s_name,customers2) <- Q2;
           c_name2 <- customers2 if c_name2 == c_name)
      yield s_name)

////////////////////////////////////////////////////////////
// Q7 - comprehension query
var Q7 =
  for (n <- Nation)
  yield (n.name,
      for ((p_name,suppliers,customers) <- Q3
           if suppliers.exists{ case (_,s_nationkey) =>
                          s_nationkey == n.nationkey } &&
              customers.forall{ case (_,c_nationkey) =>
                          c_nationkey != n.nationkey })
```

```
yield p_name)
```

List of Figures

List of Tables

Bibliography

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the Borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[2] U. A. Acar. Self-adjusting computation: (an overview). In *Proc. Workshop on Partial Evaluation and Program Manipulation*, 2009.

[3] U. A. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *Proc. POPL*, pages 309–322, 2008.

[4] U. A. Acar, G. Blelloch, R. Ley-Wild, K. Tangwongsan, and D. Turkoglu. Traceable data types for self-adjusting computation. In *Proc. PLDI*, pages 483–496, 2010.

[5] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *Proc. POPL*, pages 247–259, 2002.

[6] D. A. M. Barrington, N. Immerman, and H. Straubing. “On Uniformity within NC1”. *Journal of Computer and System Sciences*, **41**(3):274–306, 1990.

[7] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. SIGMOD Conference*, pages 61–71, 1986.

[8] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’93, pages 102–111, 1993.

[9] P. Buneman and E. K. Clemons. Efficient monitoring relational databases. *ACM Trans. Database Syst.*, 4(3):368–382, 1979.

[10] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.

[11] Y. Cai, P. G. Giarrusso, T. Rendel, and K. Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *Proc. PLDI*, pages 145–155, 2014.

[12] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 577–589, 1991.

[13] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. VLDB*, pages 577–589, 1991.

[14] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008.

[15] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.

[16] S. Chaudhuri and V. Narasayy. Tpc-d data generation with skew.

[17] J. Cheney, S. Lindley, and P. Wadler. Query shredding: Efficient relational evaluation of queries over nested multisets. In *Proc. SIGMOD*, pages 1027–1038, 2014.

[18] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, pages 469–480, 1996.

[19] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting multiple view maintenance policies. In *SIGMOD*, pages 405–416, 1997.

[20] G. Cormode and S. Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. *ACM TODS*, 30(1):249–278, 2005.

[21] K. Dimitrova, M. El-Sayed, and E. 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.

[22] M. EL-Sayed, L. Wang, L. Ding, and E. A. Rundensteiner. An algebraic approach for incremental maintenance of materialized xquery views. In *Proceedings of the 4th International Workshop on Web Information and Data Management*, WIDM '02, pages 88–91, 2002.

[23] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, Dec. 2000.

[24] J. N. Foster, R. Konuru, J. Siméon, and L. Villard. An algebraic approach to view maintenance for XQuery. In *PLAN-X 2008, Programming Language Technologies for XML*, 2008.

[25] D. Gluche, T. Grust, C. Mainberger, and M. 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.

[26] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

[27] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. SIGMOD*, pages 328–339, 1995.

[28] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: Database-supported program execution. In *Proc. SIGMOD*, pages 1063–1066, 2009.

[29] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-safe linq compilation. *Proc. VLDB Endow.*, 3(1-2):162–172, 2010.

[30] A. Gupta, D. Katiyar, and I. S. Mumick. Counting solutions to the view maintenance problem. In *Proc. Workshop on Deductive Databases, JICSLP*, 1992.

[31] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD'93*, pages 157–166, 1993.

[32] D. S. Johnson. "A Catalog of Complexity Classes". In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 1, chapter 2, pages 67–161. Elsevier Science Publishers B.V., 1990.

[33] A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. Implementing incremental view maintenance in nested data models. In *In Proceedings of the Workshop on Database Programming Languages*, pages 202–221, 1997.

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

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

[36] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.

[37] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006.

[38] P.-Å. Larson and J. Zhou. Efficient maintenance of materialized outer-join views. In *ICDE*, pages 56–65, 2007.

[39] S. K. Lellahi and V. Tannen. A calculus for collections and aggregates. In *Category Theory and Computer Science, 7th International Conference, CTCS '97, Proceedings*, pages 261–280, 1997.

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

[41] R. Ley-Wild, U. A. Acar, and M. Fluet. A cost semantics for self-adjusting computation. In *Proc. POPL*, pages 186–199, 2009.

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

[43] H. Liefke and S. B. Davidson. Specifying updates in biomedical databases. In *Proceedings. Eleventh International Conference on Scientific and Statistical Database Management*, pages 44–53, Aug 1999.

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

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

[46] Y. A. Liu. Efficiency by incrementalization: An introduction. *Higher Order Symbol. Comput.*, 13(4):289–313, Dec. 2000.

[47] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM TOPLAS*, 20(3):546–585, 1998.

[48] I. Maier and M. Odersky. Higher-order reactive programming with incremental lists. In *Proc. European Conference on Object-Oriented Programming*, ECOOP'13, pages 707–731, 2013.

[49] F. McSherry, D. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Proceedings of CIDR 2013*, January 2013.

[50] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, Sept. 2010.

[51] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[52] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, 2013.

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

[54] M. Nikolic, M. Dashti, and C. Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *Proc. SIGMOD '16*, pages 511–526, 2016.

[55] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, 2008.

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

[57] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *VLDB*, pages 802–813, 2002.

[58] J. Paredaens and D. Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Trans. Database Syst.*, 17(1):65–93, Mar. 1992.

[59] S. Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, 2008.

[60] N. Roussopoulos. An incremental access method for viewcache: Concept, algorithms, and cost analysis. *ACM Trans. Database Syst.*, 16(3):535–563, 1991.

[61] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How to roll a join: Asynchronous incremental view maintenance. In *SIGMOD*, pages 129–140, 2000.

[62] L. J. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, 1984.

[63] D. 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.

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

[65] D. Suciu and V. Tannen. Efficient compilation of high-level data parallel algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '94, pages 57–66, 1994.

[66] D. Suciu and V. Tannen. A query language for nc. *J. Comput. Syst. Sci.*, 55(2):299–321, Oct. 1997.

[67] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[68] J. 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.

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

[70] J. Van den Bussche and S. 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.

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

[72] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, 2010.

[73] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. SOSP*, pages 423–438, 2013.

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

[75] J. Zhou, P.-Å. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *VLDB*, pages 231–242, 2007.

Chemin de Chandieu, 24
1006 Lausanne
✆ *+41 (79) 862 49 70*
✉ *daniellupei@gmail.com*

Daniel Lupei

Education

Sep. 2011 - Aug. 2017
Ecole Polytechnique Fédérale de Lausanne (EPFL), *PhD in Computer Science*.
Thesis advisor Prof. Christoph Koch

Sep. 2007 - Sep. 2009
University of Toronto (UofT), *MASc in Electrical and Computer Engineering*.
Thesis advisor Prof. Cristiana Amza

Oct. 2002 - Aug. 2007
University Politehnica of Bucharest (UPB), *BASc in Computer Science and Engineering*, GPA: 9.63/10.

Core Experience

Sep. 2011 - Sep. 2017
Ecole Polytechnique Fédérale de Lausanne (EPFL), *Research Assistant*.
Proposed incremental computation techniques for query languages that operate on nested collections (as they are widely used by NoSQL systems like MongoDB, Apache Pig, Dremel or Spark). The developed approach is based on *delta processing*, for which we can statically guarantee that the derived delta query is able to update the result of the original query in response to input changes more efficiently than by re-evaluation. I implemented the approach in *SLeNDer*, a compilation framework that takes nested relational queries and generates optimized update triggers for a Spark backend. The generated code achieves up to 21.93x speedups in refreshing the results of a range of queries when compared to recomputation.

Jun. - Aug. 2015, Mar. - Jun. 2016
Microsoft Research, Redmond (USA), *Research Intern*.
Developed optimizations for pattern matching queries operating on large scale streams of complex events. The proposed techniques leverage symbolic execution and abstract interpretation to locally produce small summaries based on which the global query can be answered. Prototyped the approach on terrabytes of data on a map-reduce platform (Cosmos/Scope) and demonstrated 1000x reduction in the amount of data shuffled across the network as well as up to 50% reductions in processing and response times on production queries.

Sep. 2007 - Sep. 2009
University of Toronto (UofT), *Research Assistant*.
I worked in the area of parallel programming using Software Transactional Memory (STM) and developed *libTM*, a library implementing the STM abstraction as well as a wide range of conflict detection/resolution strategies for it. Proposed a novel optimistic conflict detection strategy with support for partial rollbacks which provides performance close to optimum regardless of the access pattern of the application. Applied *libTM* to SynQuake, a parallel massively multiplayer game server and showed improved scalability compared to a lock-based version.

Oct. 2009 - Aug. 2011
Softwin, R&D Department, Bucharest (Romania), *Software Engineer*.
Designed algorithms and data structures for natural language processing applications, including grammar checking and automatic translation. For improved accuracy, deep syntactic matching is performed based on an extensive linguistic knowledge base containing grammar rules annotated with morphological attributes.

Selected Publications

Koch, Christoph, Daniel Lupei, and Val Tannen. "Incremental View Maintenance for Collection Programming". In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '16. San Francisco, California, USA, 2016.

Koch, Christoph, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. "DBToaster: higher-order delta processing for dynamic, frequently fresh views". In: *VLDB J.* 23.2 (2014), pp. 253–278.

Lupei, Daniel, Bogdan Simion, Don Pinto, Matthew Misler, Mihai Burcea, William Krick, and Cristiana Amza. "Transactional memory support for scalable and transparent parallelization of multiplayer games". In: *Proceedings of the 5th European conference on Computer systems, EuroSys, Paris, France*. 2010.

Soundararajan, Gokul, Daniel Lupei, Saeed Ghanbari, Adrian Daniel Popescu, Jin Chen, and Cristiana Amza. "Dynamic Resource Allocation for Database Servers Running on Virtual Storage". In: *Proceedings of the 7th USENIX Conference on File and Storage Technologies, San Francisco, CA, February 24-27*. 2009.

Additional Experience

2012–2016 **Ecole Polytechnique Fédérale de Lausanne (Switzerland)**, *Teaching Assistant*.
Teaching and supervision of exercise sessions; design of exercises and assignments for courses in Advanced Databases, Algorithms, Concurrency Control and Programming Fundamentals.

2005–2006 **Outside Software, Bucharest (Romania)**, *Software Engineer*.
Developed web applications, including a Content Management System, using the .NET framework.

2004 **Microsoft Lab (UPB), Bucharest (Romania)**, *Intern*.
Created a suite of .NET client-server applications for managing a distribution network between suppliers and consumers.

Software Developing Skills

Programming Languages	Java, Scala, C/C++, C# (.NET), Python, OCaml, Haskell, JavaScript, SQL, Matlab, Octave, R
Databases	Relational: Oracle DB, MySQL / NoSQL: Apache Pig, Cosmos/Scope, BigQuery
Frameworks /Tools	Spark, Hadoop, Yarn, ASP.NET, Git, SVN, Maven, Puppet, Jenkins, Vagrant

Technical Interests

Compiler/Domain-Specific Optimizations for Expressive Analytics on Massive Datasets

Large-scale Online Processing over Streaming Data / Incremental Computation

Complex Event Pattern Matching on Time Series

Machine learning / Information Retrieval / Natural Language Processing

Automation Tools for DevOps

Academic Awards

2007–2009	Rogers Scholarship, University of Toronto
2002–2007	Study Scholarship awarded to (approx.) 10% students, UPB
2000	3rd prize at the National Mathematics Olympiad, Romania

Languages

English	Fluent	*Six years of graduate studies within English-speaking working group*
French	Conversational	*Four years of studies in Lausanne*