ALEXANDER ULRICH

QUERY FLATTENING AND THE NESTED DATA PARALLELISM PARADIGM

Query Flattening and the Nested Data Parallelism Paradigm

Dissertation
der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Diplom-Informatiker Alexander Ulrich
aus Karlsruhe

Tübingen
2018

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	21. November 2018
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Torsten Grust
2. Berichterstatter:	Prof. Dr. Klaus Ostermann

ABSTRACT

This work is based on the observation that languages for two seemingly distant domains are closely related. *Orthogonal query languages* based on comprehension syntax admit various forms of query nesting to construct nested query results and express complex predicates. Languages for *nested data parallelism* allow to nest parallel iterators and thereby admit the parallel evaluation of computations that are themselves parallel. Both kinds of languages center around the application of side-effect-free functions to each element of a collection.

The motivation for this work is the seamless integration of relational database queries with programming languages. In frameworks for *language-integrated database queries*, a host language's native collection-programming API is used to express queries. To mediate between native collection programming and relational queries, we define an expressive, orthogonal query calculus that supports nesting and order. The challenge of *query flattening* is to translate this calculus to relational queries that are restricted to flat, unordered multisets and can be executed on relational query engines. Prior solutions to this problem either support only query languages that lack in expressiveness or employ a complex, monolithic translation that is hard to comprehend and generates inefficient code that is hard to optimize.

To improve on those approaches, we draw on the similarity to nested data parallelism. Blelloch's *flattening transformation* is a static program transformation that translates nested data parallelism to flat data parallel programs over flat arrays. Based on the flattening transformation, we describe *Query Flattening*, a pipeline of small, comprehensible lowering steps that translates our nested query calculus to a bundle of relational queries. The pipeline is based on a number of well-defined intermediate languages. *Query Flattening* adopts the key concepts of the flattening transformation but is designed with the specifics of relational query processing in mind.

Based on this translation, we revisit all aspects of query flattening. *Query Flattening* is fully compositional and can translate any term of the input language. Like prior work, *Query Flattening* by itself produces inefficient code due to compositionality that is not fit for execution and requires optimization. In contrast to prior work, we show that query optimization is orthogonal to flattening and can be performed prior to flattening. We employ well-known work on logical query optimization for nested query languages and demonstrate that this body of work integrates well with *Query Flattening*.

Furthermore, we describe an improved encoding of ordered and nested collections in terms of flat, unordered relations. *Query Flattening* produces relational queries in which the effort required to maintain the non-relational semantics of the source language (order and nesting) is minimized.

A set of experiments provides evidence that *Query Flattening* handles complex, list-based queries with nested (intermediate) results well. We translate flat and nested queries and compare their runtime with hand-written as well as generated SQL queries. From the experiments we conclude that *Query Flattening* generates idiomatic relational queries that are usually as efficient as hand-written SQL code.

ZUSAMMENFASSUNG

Die vorliegende Dissertation beruht auf der Beobachtung, dass Sprachen zweier vermeintlich unterschiedlicher Domänen eng verwandt sind. *Orthogonale Abfragesprachen*, die auf *comprehension*-Syntax basieren, erlauben verschiedene Formen der Verschachtelung von Abfragen, um verschachtelte Ergebnisse sowie komplexe Prädikate auszudrücken. Sprachen für *verschachtelte Datenparallelität* erlauben das Verschachteln paralleler Iteratoren und ermöglichen es, die parallele Auswertung von Berechnungen zu beschreiben, die selbst wieder Datenparallelität ausdrücken. Beide Arten von Sprachen basieren auf der Anwendung seiteneffektfreier Funktionen auf jedes Element einer Kollektion.

Das Ziel dieser Arbeit ist es, die — idealerweise nahtlose — Einbettung von relationalen Datenbankabfragen in Programmiersprachen zu unterstützen. Frameworks für *language-integrated query* erlauben es, Datenbankabfragen mit den natürlichen Schnittstellen der Programmiersprache für Abfragen auf Kollektionen auszudrücken. Wir definieren einen ausdrucksstarken, orthogonalen Abfragekalkül, der Verschachtelung und Ordnung unterstützt, um zwischen dem Kollektions-Framework und relationalen Abfragen zu vermitteln. In diesem Zusammenhang stellt sich das Problem des *query flattening*: die Übersetzung eines derartigen verschachtelten Kalküls in relationale Abfragen, die auf flache und ungeordnete Kollektionen beschränkt sind und auf relationalen Datenbanken ausgeführt werden können. Frühere Lösungen für dieses Problem unterstützen entweder nur Abfragesrachen, deren Ausdrucksstärke nicht zufriedenstellend ist, oder basieren auf komplexen, monolithischen Übersetzungen, die ineffizienten und schwierig zu optimierenden Code erzeugen.

Wir stellen einen verbesserten Ansatz für query flattening vor, der auf der Verwandschaft zu verschachtelter Datenparallelität beruht. Die erstmals von Blelloch beschriebene *flattening transformation* ist eine statische Programmtransformation die verschachtelte Datenparallelität in flache datenparallele Programme über flachen Arrays übersetzt. Auf der Grundlage der *flattening transformation* beschreiben wir *Query Flattening*. Dabei handelt es sich um eine Abfolge von kleinen, verständlichen Übersetzungsschritten, die Ausdrücke unseres verschachtelten Kalküls in Bündel von relationalen Abfragen übersetzt. Diese Abfolge stützt sich auf eine Anzahl wohldefinierter Zwischensprachen. *Query Flattening* übernimmt die Schlüsselkonzepte der flattening transformation, bezieht aber die Besonderheiten relationaler Abfrageverarbeitung ein.

Mit dieser Übersetzung als Grundlage überarbeiten wir alle Aspekte des query flattening. Unser Ansatz *Query Flattening* ist kompositional und unterstützt beliebige Ausdrücke des verschachtelten Kalküls. Kompositionalität hat einen Preis: *Query Flattening* alleine erzeugt — wie vorherige Ansätze auch — ineffiziente Abfragen, die ohne weitere Optimierung nicht zur Ausführung geeignet sind. Im Gegensatz zu früheren Arbeiten zeigen wir, dass Abfrageoptimierung und flattening unabhängig voneinander betrachtet werden können. Wir verwenden etablierte Techniken zur logischen Optimierung von verschachtelten, orthogonalen Abfragekalkülen. Wir zeigen, dass sich diese Techniken problemlos mit *Query Flattening* zusammenfügen lassen.

Weiterhin beschreiben wir eine verbesserte Darstellung geordneter und verschachtelter Kollektionen durch flache, ungeordnete Relationen. Basierend auf diesen Darstellungen erzeugt *Query Flattening* relationale Abfragen, in denen der Aufwand für die Unterstützung der nichtrelationalen Semantik der Ausgangssprache minimiert ist.

Wir zeigen anhand von Experimenten, dass *Query Flattening* gut mit komplexen, ordnungsbasierten Abfragen mit verschachtelten (Zwischen)-Ergebnissen zurecht kommt. Wir übersetzen eine Reihe flacher und verschachtelter Abfragen und vergleichen ihre Ausführungszeit mit handgeschriebenen und generierten SQL-Abfragen. Aus diesen Experimenten ziehen wir den Schluss, dass *Query Flattening* idiomatische relationale Abfragen erzeugt. Diese sind üblicherweise nicht weniger effizient als handgeschriebene SQL-Abfragen.

ACKNOWLEDGMENTS

This work would not have succeeded without a number of people that supported me during my time as a PhD student.

First and foremost, I would like to thank my advisor Torsten Grust. From the beginning, Torsten has been a mentor and teacher in the best sense. I am grateful for the opportunity to pursue a PhD in his group. Torsten's insistence that the fields of databases and programming languages can profit from each other has been an inspiring background for my research. During my work on the thesis, Torsten provided crucial perspective and guidance whenever I got lost in details. I am also grateful that Torsten gave me the opportunity to take part in multiple Dagstuhl events and pursue a summer internship.

I would like to thank Klaus Ostermann for agreeing to be my second supervisor and for reviewing the thesis. With his perspective as a programming language researcher, Klaus provided valuable insights and advice.

Torsten's research group has been a good place to work and do research. I have benefited from working with a team of talented and motivated researchers: Dennis Butterstein, Benjamin Dietrich, George Giorgidze, Melanie Herschel, Manuel Mayr, Tobias Müller, Jan Rittinger, Tom Schreiber and Jeroen Weijers. Thank you for many interesting and fruitful discussions, and thank you for being excellent colleagues.

I would like to thank James Cheney and Sam Lindley of the University of Edinburgh for interesting discussions on query flattening and embedding of queries.

When I had left Torsten's research group, I finished writing this thesis while working at Oracle Labs. It certainly was challenging to work on the thesis next to a full-time day job, but people at Oracle Labs fully supported me in this endeavour. In particular, I would like to thank Hassan Chafi and Laurent Daynès of Oracle Labs for motivating me to finish the thesis in the final stages of writing.

CONTENTS

1 INTRODUCTION

Many high-level general-purpose programming languages provide expressive collection-programming frameworks for querying in-memory collections. These frameworks usually couple collection *combinators* that can be assembled into pipelines to filter and transform collections with *comprehension syntax*. Comprehension syntax describes the side-effect free iteration over one or multiple collections with optional predicates. Examples for collection-programming frameworks are Scala's `scala.collections` framework, Microsoft LINQ [MBB06], Haskell's lists and the Java `java.util.streams` API. Following Blelloch and Sabot [BS89; SB90], these frameworks can be considered *collection-oriented* (sub-)languages.

Evaluating queries on in-memory collections can benefit from techniques originally invented for database query processing. Deep embedding of collection queries enables relational-style optimizations on queries [Gia+13; Gia17]. Query evaluation itself can then benefit from run-time code generation, index data structures and columnar data layouts [NBV14; Nag15].

Next to in-memory collections, however, vast amounts of data are managed in actual relational database management systems and many applications rely on those. Relational database systems constitute a major part of data processing infrastructure. Relational query engines are among the most efficient solutions for large-scale data processing, with decades of investment in research and engineering. With an integration of database querying into programming languages, applications can make use of this infrastructure by offloading computations on database-resident data — *database-supported program execution* [Gru+09].

1.1 LANGUAGE-INTEGRATED QUERY

The integration of relational database queries with general-purpose programming languages has been a challenging topic. Atkinson and Buneman [AB87] observed that "Databases and programming languages have developed almost independently of one another for the past 20 years". Database query languages and general-purpose programming languages differ in data model, syntax and expressiveness. The resulting *impedance mismatch* has been described as early as 1985 by Copeland and Maier [CM84], and many others afterwards. Embedding SQL queries directly into a database application requires developers to constantly switch between two worlds, loses type safety and brings a variety of engineering problems. To quote Cheney *et al.*: "The problem is simple: two languages are more than twice as difficult to use as one language." [CLW13].

In practice, many database applications rely on *object-relational mapper* (ORM) frameworks to hide relational queries from developers and provide an object-oriented view on database-persisted relational data. However, ORM frameworks often provide leaky abstractions, can not express complex query logic and bring a variety of performance pitfalls [Yan+17; GRW08] such as the notorious $n + 1$ *Queries Problem*.

A more compelling integration of database queries and programming languages leverages the native syntax, type system and data model of the

`for (x <- xs; y <- ys if p) yield e`	(Scala)
`from x in xs from y in ys where p select e`	(C#)
`[ for x in xs for y in ys if p yield e ]`	(F#)
`[ e | x <- xs, y <- ys, p ]`	(Haskell)

Table 1: Comprehension syntax in various programming languages.

host language: *language-integrated database query* allows to formulate database queries using the native collection framework (*e.g.* LINQ) of the host language. Queries are written using the same combination of collection combinators and comprehension syntax used for in-memory collections.

The usefulness of comprehension syntax for database querying has been recognized by Trinder [Tri91] as well as Buneman *et al.* [Bun+94]. Comprehensions readily express query operations like projections and joins and are closely related to query calculi. Table 1 shows examples of comprehension syntax in various languages.

Language-integrated database query is widely used. Microsoft's LINQ framework provides language-integrated database query for a multitude of programming languages (*e.g.* C#, F#). *Slick* [Slick] embeds database queries in Scala applications based on the native Scala collection framework. Other examples include the research language Links [Coo+06], Wong's Kleisli system [Won00; Won94] and Database-Supported Haskell [Gio+11a].

1.2 BROKEN PROMISES

Ideally, use of language-integrated database query should bring no surprises compared to querying in-memory collections. From the perspective of a programmer, the integration should be *seamless*.

In reality, however, language-integrated database query facilities regularly break the promise of a seamless integration. Programmers are used to in-memory collections that can be arbitrarily *nested* (*e.g.* lists of lists) while relational systems support only flat relations. Nested collections are naturally associated with *nested queries* to filter and transform nested collections at any level. In LINQ, queries with nested results either lead to a runtime error or are executed via an inefficient *avalanche* of individual database queries [GRS10]. Also, in-memory collections can be *ordered* (*i.e.* lists, sequences) while SQL is based on unordered multisets. The native collection type of Links, for example, are ordered lists. Links database queries using the same syntax and types as for lists, however, only provide unordered multiset semantics that directly maps to SQL.

Broken promises in query embedding can cause actual breakage. The Scala snippet in Figure 1 expresses grouping and aggregation using *Slick*. Grouping with aggregation is expressed in two steps. Grouping is based on the regular Scala `groupBy` operator that returns a nested collection of groups (expression `q`, Line 1). The groups are aggregated in a second step by applying aggregation combinators (`length`, `avg`) to each group (expression `q2`, Line 6). Expression `q2` utilizes `q` and can be translated to a SQL query by *Slick*. Expression `q`, however, is a valid, type-correct collection expression in its own right and could be executed without problems on an in-memory collection. Due to the nested result type of `q`, however, *Slick* is not able to translate it to a SQL query and generates a runtime error.

```
val q = (for {
  c <- coffees
  s <- c.supplier
} yield (c, s)).groupBy(_._1.supID)

val q2 = q.map { case (supID, css) =>
  (supID, css.length, css.map(_._1.price).avg)
}
```

Figure 1: Grouping and aggregation in *Slick*. Example copied from the *Slick* 3.2.3 documentation [Slick].

Seamlessly language-integrated database queries are the motivating background for this thesis. We argue that the data model of nested and ordered collections along with support for nested iteration should be extended to the subset of a programming language that can be used for database queries. In Section 1.3, we motivate this opinion based on an example of language-integrated database queries in Haskell.

We incur, however, the challenge of *query flattening*: we need a way to translate nested queries over nested and ordered collections into efficient relational queries that are restricted to flat and unordered relations. In Chapter 2 we survey prior work on query flattening and analyze its shortcomings. In the remainder of this thesis, we then go on to describe an approach to query flattening that improves over the state of the art.

1.3 CASE STUDY: DATABASE-SUPPORTED HASKELL

Haskell is a purely-functional, statically typed general-purpose programming language with lazy evaluation [Mar10]. Database-Supported Haskell (DSH) [Gio+11a] integrates relational database queries in Haskell based on list combinators and list comprehensions[1]. Queries written in the DSH subset of Haskell mimic regular Haskell list programs both in syntax and types. DSH admits a rich data model in which query results can be arbitrary combinations of lists, tuples and atomic values — in particular, query results may feature nested lists. DSH queries are type-checked just as regular Haskell code: while `e :: [a]` denotes an expression `e` of type `[a]` (list of `a`), expression `q :: Q [a]` is a query that returns a value of type `[a]` when executed. A query of type `Q [a]` can be executed by the DSH runtime on a relational database to obtain the resulting Haskell value of type `[a]`[2]. DSH faithfully preserves the order semantics of the Haskell list combinators.

DSH imposes a list view on relational database tables and provides access to tables in the form of lists of records. The following excerpt of a record type definition maps to the schema of the customer table in the TPC-H schema [TPC-H]:

```
data Customer = C { c_custkey :: Integer, c_name :: Text, ... }
```

1 DSH as described by Giorgidze *et al.* uses quasi-quoting to overload list comprehension syntax for database queries. In this thesis, we describe a version of DSH that utilizes monad comprehensions [Gio+11b] instead.

2 For the sake of readability, we slightly simplify types in our discussion of DSH. The actual DSH implementation uses type classes to restrict type variables to those Haskell types — lists, tuples and atomic values — that are supported by DSH

```
ordersOf :: Q Customer -> Q [Order]
ordersOf c = filter (\o -> o_custkeyQ o == c_custkeyQ c) orders

orderVolume :: Q [(Text, Decimal)]
orderVolume =
  [ (c_nameQ c, sum [ o_totalpriceQ o | o <- ordersOf c])
  | c <- customers ]
```

Figure 2: A simple DSH query: compute the total of each customers' orders.

`map`	`:: (Q a -> Q b)-> Q [a] -> Q [b]`
`concatMap`	`:: (Q a -> Q [b])-> Q [a] -> Q [b]`
`filter`	`:: (Q a -> Q Bool)-> Q [a] -> Q [a]`
`sortWith`	`:: Ord b => (Q a -> Q b)-> Q [a] -> Q [a]`
`groupWith`	`:: Eq b => (Q a -> Q b)-> Q [a] -> Q [(b,[a])]`
`concat`	`:: Q [[a]] -> Q [a]`
`nub`	`:: Eq a => Q [a] -> Q [a]`
`elem`	`:: Eq a => Q a -> Q [a] -> Q Bool`
`enum`	`:: Q [a] -> Q [(a,Integer)]`
`take`	`:: Q Integer -> Q [a] -> Q [a]`
`drop`	`:: Q Integer -> Q [a] -> Q [a]`
`zip`	`:: Q [a] -> Q [b] -> Q [(a,b)]`
`length`	`:: Q [a] -> Q Integer`
`sum`	`:: Num a => Q [a] -> Q a`
`maximum`	`:: Ord a => Q [a] -> Q a`
`minimum`	`:: Ord a => Q [a] -> Q a`
`and`	`:: Q [Bool] -> Q Bool`
`or`	`:: Q [Bool] -> Q Bool`
`mins`	`:: Num a => Q [a] -> Q [a]`

Table 2: Haskell list combinators supported by DSH (excerpt)

In queries, the database-resident list of customers is denoted by the expression `customers` of type `Q [Customer]` (sorted in primary-key order). DSH automatically provides query versions of the record field selectors: While `c_custkey` is of type `Customer -> Integer`, its query equivalent `c_custkeyQ` is of type `Q Customer -> Q Integer`.

Figure 2 shows a basic query in DSH based on the TPC-H schema that answers the following question: What is the total price of all orders of each customer? The one-to-many relationship between customers and orders is expressed by the function `ordersOf`: Given one customer, it computes all corresponding order records. For each order, `orderVolume` relies on `ordersOf` to fetch the corresponding orders and aggregates them. The query is written in the typical style of Haskell list programs by combining list combinators (`sum`, `filter`) and list comprehensions. Indeed, the only artifacts that mark the program as a DSH query to be executed on a database system are the type annotations `Q`. The list combinators are DSH versions that mimic their regular Haskell counterparts. Table 2 lists an excerpt of combinators supported by DSH.

```
-- margin ≙ current value - minimum value up to now
margins :: (Ord a, Num a) => Q [a] -> Q [a]
margins xs = [ x - y | (x,y) <- zip xs (mins xs) ]

-- our profit is the maximum margin obtainable
profit :: (Ord a, Num a) => Q [a] -> Q a
profit xs = maximum (margins xs)

-- best profit obtainable for stock on given date
bestProfit :: Text -> Date -> Q [Trade] -> Q Double
bestProfit stock date trades =
  profit [ price t | t <- sortWith ts trades,
                     id t  == toQ stock,
                     day t == toQ date ]
```

Figure 3: Best profit obtainable if we buy, then sell stock. `bestProfit "ACME" "10/20/2014" trades` yields `6.0`.

In the following, we discuss more closely query formulation in DSH. Through a series of examples we identify aspects that – in our opinion – make a subset of a programming language practically usable as a database query language.

1.3.1 *Ordered Data Model*

The data model of real-world relational database systems is centered around unordered multisets. In contrast, the collection type in functional programming languages is often ordered (for example, Haskell and Links both prominently feature *lists*). To preserve the semantics of the host language, a language-integrated query system should maintain the order semantics on the database.

In an orthogonal collection programming framework, sorting is expressed with combinators (*e.g.* Haskell's `sortWith`) that can be used without restrictions in a query. In a query language based on unordered collections, sorting can not be used in arbitrary locations. Sorting can only be expressed as post-processing on the top-level of queries, in the style of SQL's **`ORDER BY`** clause. Thus, making sorting available as a first-class operation in an orthogonal language requires ordered collections.

In addition, an ordered data model is useful to formulate complex query logic in inherently order-aware domains. To make this case, we adopt an example query by Lerner and Shasha [LS03] based on a time series of stock trades. For a given stock on a specific day, we want to answer the following question: What is the maximum profit that we can obtain by first buying the stock and then selling it later on the same day? We answer this question with the list-processing DSH query `bestProfit` in Figure 3.

Function `bestProfit` sorts trading records sorted by their timestamp — expression `trades` references a database-resident table (Line 12). Note that this is the only point in the query that explicitly specifies the order of data. The remainder of the query is based on list semantics and preserves the order of input lists. For example, in `bestProfit`, the comprehension that filters the sorted trading time series based on stock and day will preserve its order.

Based on the ordered representation of the trading time series, the query is straightforward to implement with two helper functions. For each trad-

ing record, we compute the minimum price of all *preceding* trades, and the difference between both prices is the *margin* obtainable by selling at this point in time (function `margins`). DSH provides the combinator `mins` that computes a *running minimum* on its input list. For example, `mins [6,5,9,3]` evaluates to `[6,5,5,3]`. The maximum of the margins for all records is the best profit obtainable at the given day (`profit`). Based on `mins` and `margins`, computation of the actual maximum profit is easy (Line 7).

Lerner and Shasha [LS03] argue that an ordered data model and a set of corresponding primitives admits a formulation of queries on time series and other inherently ordered data that is both simple and compact. The DSH query formulation in Figure 3 mirrors the formulation of the same query in their ordered query language *AQuery*. In contrast, query languages that add limited order-aware operators to an unordered data model (*e.g.* SQL:2003 window functions) tend to be complex to use.

1.3.2 *Abstraction Over Queries, Building Tools*

Functional programming languages emphasize a style of programming in which small functions that perform well-defined, isolated tasks are combined. DSH extends this style to database queries: DSH's deep embedding approach allows to freely combine individual Haskell functions into queries. This property is exploited in query `orderVolume` (Figure 2): Fetching a customer's orders is delegated to function `ordersOf`. DSH allows to abstract over arbitrary parts of a query. Crucially, abstraction in DSH queries comes for free. An invocation of `ordersOf` in the enclosing iteration of `orderVolume` does *not* cause DSH to issue a separate backend query. Instead, DSH inlines function applications and translates `orderVolume` as a whole.

Query abstraction allows to build domain-specific libraries of reusable building blocks from which complex queries can be assembled. Building blocks like `ordersOf` can be reused in other queries on the TPC-H schema. In Figure 4 we reuse `ordersOf` in a DSH formulation of TPC-H Q22 that identifies potential customers in specific areas. Customers are deemed interesting if their account balance is above average and they do not have any open orders. For each area, the query asks for the summary account balance of the corresponding potential customers.

The DSH query splits this problem into easily digestible parts. Computing the average account balance of a list of customers is implemented in function `avgBalance` (Line 2). In the SQL formulation of Q22, computing the average account balance is entwined with the area-based selection of customers — here, these tasks can be implemented independently. Filtering potential customers is delegated to `potentialCustomers` which itself relies on other building blocks (`avgBalance`, `ordersOf`) to implement the complex predicates. Filtering customers based on their areas is implemented in function `livesIn`. The main function `q22` merely has to combine these abstractions to select eligible customers and then compute aggregates to prepare the actual report.

Abstraction in queries is not only useful to provide domain-specific tools. We can build more general abstractions that are not tied to any particular domain. True to the functional nature of Haskell, we can build higher-order abstractions. Abstraction `topK` provides a means to express *top-k* queries that select the `k` elements with the highest rank.

```
-- Top k elements in xs based on ordering criterion f
topK :: Ord b => Q Integer -> (Q a -> Q b) -> Q [a] -> Q [a]
```

```
-- average account balance of the customers cs
avgBalance :: Q [Customer] -> Q Double
avgBalance cs = avg [ c_acctbal c | c <- cs, c_acctbal c > 0.0 ]

-- high potentials among the customers cs
potentialCustomers :: Q [Customer] -> Q [Customer]
potentialCustomers cs =
  [ c | c <- cs, c_acctbal c > avgBalance cs, empty (ordersOf c) ]

-- country code (phone number prefix) for customer c
countryCodeOf :: Q Customer -> Q Text
countryCodeOf c = subString 2 (c_phone c)

-- does customer c live in any of the given countries?
livesIn :: Q Customer -> [Text] -> Q Bool
livesIn c countries = countryCodeOf c `elem` toQ countries

-- TPC-H query Q22
q22 :: [Text] -> Q [(Text, Integer, Double)]
q22 countries =
  sortWith (\(country, _, _) -> country)
    [ (country, length custs, sum (map c_acctbal custs)) |
      (country, custs) <- groupWith countryCodeOf pots ]
  where
    pots = potentialCustomers [ c | c <- customers,
                                    c `livesIn` countries ]
```

Figure 4: A Haskell formulation of TPC-H query *Q22*.

```
topK k f xs = take k $ reverse $ sortWith f xs
```

The ordering criterion that determines the rank is provided as an argument `f` to the generic `topK` combinator. `topK` sorts its input list in the order imposed by the ordering criterion and then returns a prefix of this list. Note that the ordering criterion is not limited to simple scalar expressions. *Any* DSH abstraction that fits the generic type of argument `f` can be employed as an ordering criterion. For example, `topK` can be used to retrieve the 10 customers with the most orders:

```
topK 10 (length . ordersOf) customers
```

In a recent publication [UG15], we have described this style of querying as *layered*. Complex queries are assembled from layers of abstractions, consisting of domain-specific building blocks and generic tools. These two layers

domain	`margins` `ordersOf` `fromNation` ...
generic	`topK` `groupagg` `mins` `groupWith` `sortWith` `zip` `concat` `map` `elem` `reverse` `take/drop` `head` `!!`
primitive	`[·|·]` `enum` aggregates (`sum`, `length`, `or`, ...) `++` `sort` `nub` `empty` simple expressions (+, ==, ...)

Figure 5: Three layers of query abstractions. Operations in upper layers are definable in terms of those on lower layers (Figure adapted from [UG15]).

```
-- Does customer c have nationality nation?
fromNation :: Q Customer -> Q Text -> Q Bool
fromNation c nation =
  or [ n_nationkey n == c_nationkey c && n_name n == nation |
       n <- nations ]

-- When did customers from nation place their k most expensive orders?
topOrders :: Q Integer -> Q Text -> Q [(Text, [Day])]
topOrders k nation =
  [ (c_name c,
     [ o_orderdate o | o <- topK k o_totalprice (ordersOf c) ]) |
    c <- customers,
    c `fromNation` nation ]
```

Figure 6: DSH query with a nested result.

are themselves constructed from a small number of primitives which are provided by the embedded query language (Figure 5).

Formulating queries in this style brings a requirement, though: our query language needs to be fully *orthogonal*. Queries obtained as a composition of arbitrary abstractions can not be guaranteed to fit in any restricted syntactical skeleton like `select-from-where` blocks in SQL. All query constructs can be used independently of their context. Concretely, in DSH any type-correct term of type `Q a` is a valid query that can be executed.

1.3.3 *Queries With Nested Results*

In the subset of Haskell we consider here, type constructors $[\cdot]$ for lists and $(\cdot,\cdot)$ for tuples can be nested arbitrarily. In particular, lists can be nested. In order to properly integrate queries, we have to preserve this property in the embedded query language.

Layered queries naturally lead to nested intermediate results, even if the queries' result type is flat. The following query computes the number of orders placed by each customer.

```
orderCount :: Q [(Customer, Integer)]
orderCount = zip customers (map length $ map ordersOf customers)
```

The expression `map ordersOf customers` has the nested type `Q [[Order]]`. Although the inner lists are subsequently aggregated to form a flat result, the embedded query language has to be able to represent nested data.

Nested lists may not only occur as intermediate data but also as query results. A common pattern in database programming is the following: For each element of an outer collection, obtain a number of related elements from another inner collection. A naive implementation of this pattern results in a separate backend query being issued to the database for each element of the outer collection. The resulting query avalanche [GRS10] harrows users of object-relational mappers.

With a nested data model, on the other hand, we can describe this pattern as a single query with a nested result as shown in query `topOrders` in Figure 6: for each customer belonging to a given nation, it computes the dates on which the customer's most expensive orders have been placed. Again the query is implemented as a composition of domain-specific and generic abstractions. First, only those customers are retained that are from a given nation. Abstraction `fromNation` expresses the existential quantification us-

```
-- Line items of order o
itemsOf :: Q Order -> Q [Lineitem]
itemsOf o = [ l | l <- lineitems, l_orderkey l == o_orderkey o ]

-- Line items of order o and how many days they have been delayed
delaysOf :: Q Order -> Q [(Lineitem, Integer)]
delaysOf o = [ (l, o_orderdate o - l_shipdate l) | l <- itemsOf o ]

-- Orders whose line items are delayed more than 5 days on average
shippingDelay :: Q [(Integer, [LineItem]]
shippingDelay = [ (o_orderkey o, map fst (sortWith snd ds)) |
                  o <- orders, let ds = delaysOf o,
                  sum (map snd ds) > 5 ]
```

Figure 7: DSH query: compute shipping delays.

ing the Boolean aggregate `or`. For each relevant customer, the implementation employs `ordersOf` to fetch all corresponding orders. As we are not interested in all orders of a customer, we use generic abstraction `topK` to reduce them to the `k` most expensive ones.

Figure 7 lists another query with a nested result on the TPC-H schema. It computes the lineitems of an order `o` together with the number of days that shipment of each lineitem has been delayed. The delay is here understood as the difference between the order date and actual shipping date of a lineitem. The query only considers those orders for which the total delay exceeds a certain threshold. The query demonstrates that DSH allows sorting at an arbitrary place in a query. For each order, the list of corresponding lineitems is sorted individually. In the remainder of this thesis we use query `shippingDelay` as a running example.

1.4 A LIST-BASED QUERY LANGUAGE

To recapitulate, we argue in Section 1.3 that a language-integrated query system should have the following properties:

- A data model that allows intermediate results as well as final query results to be nested.
- A combination of an ordered collection type and the means to express order-aware operations (*e.g.* sorting, enumeration, *top-k*).
- The embedded query language should be expressive and include functionality like grouping, aggregation and duplicate elimination.
- The query language should be *orthogonal* not only in types but also in terms [BBW92] and allow arbitrary combinations of combinators and comprehensions.

Implementing a language-integrated query system brings two largely orthogonal tasks: (1) language embedding and (2) translation to SQL. Queries need to be integrated with the syntax and type system of the host language (1) and reified into a form that is subsequently translated to SQL queries (2). In this work, we focus on task (2). We define the query language $\mathcal{CL}$ that has the properties listed above. $\mathcal{CL}$ is designed as an intermediate representation for a language-integrated query facility such as DSH. The DSH

$$
\begin{aligned}
t &::= \langle \text{table name} \rangle \\
v &::= \langle \textit{scalar} \text{ literal value} \rangle \\
x, y &::= \langle \text{variable name} \rangle \\
\ell &::= \langle \text{record field label} \rangle
\end{aligned}
$$

Expressions, Comprehension Syntax

$$
\begin{aligned}
e ::= {} & v \mid x \mid \texttt{[}v, \ldots, v\texttt{]} \mid \texttt{table(t)} \mid \texttt{let } x = e \texttt{ in } e \\
& \mid p\ e \cdots e \mid \texttt{reduce\{}s, s\texttt{\}}\ e \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \\
& \mid \texttt{[} e \texttt{ | } x \leftarrow e, qs \texttt{]} \\
qs ::= {} & x \leftarrow e, qs \mid e, qs \mid \varepsilon
\end{aligned}
$$

Built-In Functions and Operators

$$
\begin{aligned}
p ::= {} & \texttt{sort} \mid \texttt{\#} \mid \texttt{concat} \mid \texttt{distinct} \mid \texttt{group} \mid \texttt{append} \\
& \mid \texttt{sng} \mid \langle \ell = \_, \ldots, \ell = \_ \rangle \mid \texttt{c(\_,...,\_)} \mid \_.\ell
\end{aligned}
$$

Scalar Expressions

$$
\begin{aligned}
s ::= {} & x \mid v \mid \texttt{c(}s, \ldots, s\texttt{)} \mid s.\ell \mid \langle \ell = s, \ldots, \ell = s \rangle \\
& \mid \texttt{if } s \texttt{ then } s \texttt{ else } s \mid \lambda x \cdots x.s \mid s\ s\ \texttt{...}\ s
\end{aligned}
$$

Figure 8: Syntax of query language $\mathcal{CL}$.

frontend translates a reified DSH query into $\mathcal{CL}$. $\mathcal{CL}$ queries are then further translated to SQL queries ready to be executed on a database.

$\mathcal{CL}$ is an orthogonal query calculus centered around comprehensions and combinators. $\mathcal{CL}$ is similar to query calculi that have been defined as a theoretical basis for complex-object query languages like OQL [Bun+95; Gru99; FMoo]. It operates on arbitrarily nested lists and records. Our running example, the DSH query `shippingDelay` (Figure 7), can be expressed in $\mathcal{CL}$ as follows:

```
[ ⟨o, sort [ ⟨l, l.sd - o.od⟩ | l ← ls, l.ok = o.ok ]⟩
| o ← os                                                    (Q1)
, 5 < sum [ l.sd - o.od | l ← ls, l.ok = o.ok ] ]
```

Just as DSH queries, Query Q1 centers around comprehensions and uses additional list combinators. Combinator `sort` takes a list of pairs whose second component specifies the sorting key of each element. In the remainder of this thesis, we use some convenient shortcuts for the TPC-H schema. We write `os` for `table(orders)` and `ls` for `table(lineitem)`. Additionally, we use the shortcuts `od` and `sd` for the record labels `o_orderdate` and `l_shipdate`. Depending on the context we write `ok` for the record labels `o_orderkey` and `l_orderkey`.

The syntax of $\mathcal{CL}$ terms is defined in Figure 8. List comprehensions `[ e | qs ]` are the central construct that expresses side-effect free iteration over multiple lists as well as filtering of lists. We call expression e the *head* expression and qs the *qualifiers*. Qualifiers are a non-empty sequence of *generators* $x \leftarrow e$ and *guard* expressions[3].

Base operators `c(_,...,_)` include arithmetic, boolean and comparison operators as well as `max` and `min` which return the larger and smaller of two values, respectively. We require equality and order on all scalar types

3 Note that the grammar includes a dangling comma at the end of the qualifier sequence which we ignore.

including records. Order on records is defined as the lexicographic order on record labels. We write pairs $\langle \_, \_ \rangle$ as a shortcut for the record constructor $\langle \texttt{1=}\_\texttt{, 2=}\_ \rangle$. Constants are either base literal values, literal lists of base values or table references `table(t)`. We assume tables to be flat multisets and interpret them as flat lists of records in the order of the primary key.

Combinator $\texttt{reduce\{}s_z\texttt{, }s_f\texttt{\}}$ folds a list into a scalar value and subsumes aggregation functions. Scalar expression s_z defines the initial value and s_f defines the folding function. Note that we do not define a particular order for the fold. Folding functions are required to be associative. Combinator $\texttt{reduce\{}s_z\texttt{, }s_f\texttt{\}}$ may compute multiple aggregates at once. In the following example, a list is folded into a pair of the element sum and the length.

$$\texttt{reduce\{}\langle \texttt{0, 0} \rangle\texttt{, }\lambda \texttt{a x.}\langle \texttt{a.1 + x, a.2 + 1} \rangle\texttt{\} [4, 5, 6]} \equiv \langle \texttt{15, 3} \rangle$$

In examples, we sometimes use concrete aggregate combinators like `sum` that map to `reduce{}`.

The grammar includes a syntactic category s of *scalar* expressions. These are restricted to atomic and record operations but do also include lambda abstraction and application. For now, scalar expressions are used exclusively as fixed arguments to $\texttt{reduce\{}s_z\texttt{, }s_f\texttt{\}}$. Actual expressions of $\mathcal{CL}$ do *not* include functions. In scalar expressions we use some usual notational shortcuts for lambda abstractions:

$$\begin{aligned} s_1 \circ s_2 &\equiv \lambda \texttt{x.}s_1\ (s_2\ \texttt{x}) \\ \pi_\ell &\equiv \lambda \texttt{x.x.}\ell \\ s_1 \oslash s_2 &\equiv \lambda \texttt{x y.}s_1\ \texttt{x}\ (s_2\ \texttt{y}) \\ s_1 \parallel s_2 &\equiv \lambda \texttt{x.}\langle s_1\ \texttt{x.1}, s_2\ \texttt{x.2} \rangle \end{aligned}$$

1.4.1 *Static Semantics*

Types in $\mathcal{CL}$ have the following form. We let τ range over general types — arbitrary combinations of lists and records — and π over atomic *base types*. Additionally, we let δ range over *scalar types* that only include nested records of base types.

Base Types

$$\pi ::= \texttt{Int} \mid \texttt{Text} \mid \texttt{Bool} \mid \texttt{Date} \mid \langle\rangle$$

Types

$$\tau ::= \pi \mid \texttt{[}\tau\texttt{]} \mid \langle \ell\texttt{:}\tau\texttt{,...,}\ell\texttt{:}\tau \rangle$$

Scalar Types

$$\delta, \alpha, \beta, \gamma ::= \pi \mid \langle \ell\texttt{:}\delta\texttt{,...,}\ell\texttt{:}\delta \rangle$$

We say that a type is *nested* if the element type of a list type constructor contains further nested list type constructors. Otherwise, a type is *flat*.

We define the static semantics of $\mathcal{CL}$ using the (standard) typing rules in Figure 9. We let Γ and Δ range over *type environments* that map variables to types. The inference rules in Figure 9a define the typing relation for terms. The language's central construct is the list comprehension, for which we adopt the typing rules from Peyton Jones and Wadler [PW07]. In a comprehension $\texttt{[} e \texttt{ | } qs\texttt{, } q\texttt{, } qs' \texttt{]}$ qualifier q is type-checked in a type environment extended with bindings for all preceding qualifiers qs. Likewise, the head expression e is type-checked in a type environment extended with

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \mathcal{CL}\text{-TY-VAR} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \mathcal{CL}\text{-TY-LET}$$

$$\frac{\Sigma(t) = \langle \ell_i{:}\pi_i \rangle_{i=1}^n}{\Gamma \vdash \texttt{table(t)} : [\langle \ell_i{:}\pi_i \rangle_{i=1}^n]}\ \mathcal{CL}\text{-TY-TABLE} \qquad \frac{\vdash v : \pi}{\Gamma \vdash v : \pi}\ \mathcal{CL}\text{-TY-LIT-BASE} \qquad \frac{[\vdash v_i : \pi]_{i=1}^n}{\Gamma \vdash [v_1, \ldots, v_n] : [\pi]}\ \mathcal{CL}\text{-TY-LIT-LIST}$$

$$\frac{\Sigma(p) = \tau_1 \to \cdots \to \tau_n \to \tau \qquad [\Gamma \vdash e_i : \tau_i]_{i=1}^n}{\Gamma \vdash p\ e_1\ \cdots\ e_n : \tau}\ \mathcal{CL}\text{-TY-OP}$$

$$\frac{\Gamma \vdash e : [\tau] \qquad \vdash s_z : \delta_a \qquad \vdash s_f : \delta_a \to \tau \to \delta_a}{\Gamma \vdash \texttt{reduce}\{s_z, s_f\}\ e : \delta_a}\ \mathcal{CL}\text{-TY-REDUCE}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}\ \mathcal{CL}\text{-TY-IF}$$

$$\frac{\Gamma \vdash e_1 : [\tau_1] \qquad \Gamma, x{:}\tau_1 \vdash qs \Rightarrow \Delta \qquad \Gamma, x{:}\tau_1, \Delta \vdash e : \tau}{\Gamma \vdash [\ e \mid x \leftarrow e_1, qs\ ] : [\tau]}\ \mathcal{CL}\text{-TY-COMPREHENSION}$$

(a) Typing of terms.

$$\frac{\Gamma \vdash e : [\tau] \qquad \Gamma, x{:}\tau \vdash qs \Rightarrow \Delta}{\Gamma \vdash x \leftarrow e, qs \Rightarrow \{x{:}\tau\} \cup \Delta}\ \mathcal{CL}\text{-TY-GENS} \qquad \frac{\Gamma \vdash e : [\tau]}{\Gamma \vdash x \leftarrow e \Rightarrow \{x{:}\tau\}}\ \mathcal{CL}\text{-TY-GEN}$$

$$\frac{\Gamma \vdash e : \texttt{Bool} \qquad \Gamma \vdash qs \Rightarrow \Delta}{\Gamma \vdash e, qs \Rightarrow \Delta}\ \mathcal{CL}\text{-TY-GUARDS} \qquad \frac{\Gamma \vdash e : \texttt{Bool}}{\Gamma \vdash e \Rightarrow \varnothing}\ \mathcal{CL}\text{-TY-GUARD}$$

(b) Typing of comprehension qualifiers.

Figure 9: Typing rules for $\mathcal{CL}$.

bindings for all qualifiers. Figure 9b defines a typing relation for qualifier lists that constructs the type environment for subsequent qualifiers as well as the head expression. Judgment $\Gamma \vdash qs \Rightarrow \Delta$ indicates that in type environment Γ qualifiers qs lead to type bindings in Δ. In typing rules we let $\Sigma(t)$ denote the record type of table t. Also, $\Phi(t)$ denotes the scalar type of the primary key of table t. We omit the typing rules for scalar expressions s which are the standard typing rules for the simply-typed lambda calculus with records and base types [Pie02].

Table 3 lists the type signatures of combinators provided by $\mathcal{CL}$. List combinators like `sort` are restricted in the type of their input arguments. The second pair component that specifies the sorting key must have a scalar type. This ensures that we can map it to relational operators later.

As a shortcut, we write `Order` and `Lineitem` for the element record type of tables `table(orders)` and `table(lineitem)`. Then, Query Q1 has type `[⟨Order,[Lineitem]⟩]`.

$c(\_, \ldots, \_)$	:	$\delta_1 \to \cdots \to \delta_n \to \delta$
$\langle \ell_1 = \_, \ldots, \ell_n = \_ \rangle$	:	$\tau_1 \to \cdots \to \tau_n \to \langle \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \rangle$
$\_.\ell$	:	$\langle \ldots, \ell : \tau, \ldots \rangle \to \tau$
`sort`	:	$[\langle \tau, \delta \rangle] \to [\tau]$
`number`	:	$[\tau] \to [\langle \tau, \texttt{Int} \rangle]$
`concat`	:	$[[\tau]] \to [\tau]$
`group`	:	$[\langle \tau, \delta \rangle] \to [\langle \delta, [\tau] \rangle]$
`distinct`	:	$[\delta] \to [\delta]$
`sng`	:	$\tau \to [\tau]$
`append`	:	$[\tau] \to [\tau] \to [\tau]$

Table 3: Type signatures of builtin $\mathcal{CL}$ combinators.

1.4.2 *Frontend Translation*

With DSH we advocate constructing complex queries by assembling small, well-contained building blocks, including higher-order abstractions. At the same time, built-in query combinators like `sortWith` that are provided by DSH are higher-order functions as well. Our query language $\mathcal{CL}$, on the other hand, does not have a notion of functions. Built-in combinators like `sort` don't take functional arguments and are not first-class values themselves. They can only be fully applied to form expressions.

This is not actually a contradiction. In this thesis, we focus on the translation of nested, ordered $\mathcal{CL}$ queries into flat relational queries. The embedding of $\mathcal{CL}$ into languages with first-class functions is an orthogonal topic that has been discussed in prior work. Cheney *et al.* [CLW13], for instance, provide a theory of language-integrated queries based on quotation. In their work, queries are normalized and all abstractions are inlined. At the point at which queries are actually translated to relational code, the query is first-order and no abstractions remain. Subsequent work [CLW14a] of the same authors on nested queries follows the same pattern: only a first-order query language without abstractions is considered for translation to relational queries (see also Chapter 2). DSH uses a similar but less principled approach.

This restriction is only natural: relational query languages (*i.e.* SQL) have no notion of functional abstraction for good reasons. Hence, inlining functions is necessary. In the work of Cheney *et al.*, normalization provides the guarantee that all functions are inlined. Even if that guarantee can not be obtained, queries with residual function applications can still be rewritten into first-order variants before being translated to relational queries [GU13]. In this work, we have decided to consider only a first-order query language without function application in order to focus on the core aspects of query translation.

2 RELATED WORK ON QUERY FLATTENING

Query flattening translates a query in a nested query language into a number of flat queries in a flat query language. Hence, query flattening enables the execution of queries with nested results and nested intermediate results on flat relational query engines. The problem of query flattening is not new. In this chapter, we survey literature on query flattening.

Flat-to-flat queries in a nested query language can still produce nested intermediate results. It is well-known that for a large class of queries, such intermediate results can be eliminated such that only one flat query needs to be translated (Section 2.1.2). However, we are interested in translations that can deal with *flat-to-nested* queries as well.

One strategy to execute nested queries of the form $[\ q_2\ |\ x \leftarrow q_1\ ]$ is to execute the query q_1 first and then execute one instance of query q_2 for each element in the result of q_1 — assuming that q_1 and q_2 are both flat. However, the number of flat queries issued depends on the cardinality of the result of q_1 and can quickly overwhelm a database system if that cardinality is large. Generating such a *query avalanche* is an escape hatch for systems that can not deal properly with nested queries and is known to be inefficient [GRS10]. We are only interested in translations for which the number of flat queries that is generated is bounded and can be statically determined. This excludes Wong's Kleisli system [Wono0], for example. For all query flattening translations we survey in this chapter, the number of flat queries is determined statically by the type of the query result. Our example Query Q1 features two list type constructors in its result type `[⟨Order,[Lineitem]⟩]` and can be translated to two flat queries.

2.1 FOUNDATIONS OF QUERY FLATTENING

We first review descriptions of query flattening in database theory.

2.1.1 *Data Flattening*

Multiple authors formalize encodings of nested relations into multiple flat relations. Roth *et al.* [RKS88] define the *Partitioned Normal Form* (PNF) for nested relations. A nested relation is in PNF if its atomic attributes form a super key and its complex attributes are in PNF as well. *Partitioned Normal Form* is the basis for encodings of nested relations into flat relations.

Abiteboul and Bidoit [AB86] as well as Hulin [Hul90] define a flat encoding of nested relations that are in PNF using flat relations. Both use flat relations as an intermediate representation for schema transformations. We

$$r_1 = \{\texttt{"Jones"}, \texttt{"Smith"}\}$$

$$r_2 = \{\langle \texttt{"Jones"}, \texttt{"math"} \rangle, \langle \texttt{"Jones"}, \texttt{"science"} \rangle, \langle \texttt{"Jones"}, \texttt{"physics"} \rangle, \langle \texttt{"Smith"}, \texttt{"physics"} \rangle\}$$

$$r_3 = \{\langle \texttt{"Jones"}, \texttt{"math"}, 1977, \texttt{"Russel"} \rangle, \langle \texttt{"Jones"}, \texttt{"math"}, 1978, \texttt{"Russel"} \rangle, \langle \texttt{"Jones"}, \texttt{"math"}, 1978, \texttt{"Doolittle"} \rangle, \langle \texttt{"Jones"}, \texttt{"physics"}, 1977, \texttt{"Martin"} \rangle, \langle \texttt{"Smith"}, \texttt{"physics"}, 1978, \texttt{"Anderson"} \rangle\}$$

Figure 10: Flat representation of nested set r according to Hulin [Hul90].

adopt the following example from Hulin [Hul90] using set notation. Consider the following nested set r of type $\{\langle \texttt{Text}, \{\langle \texttt{Text}, \{\langle \texttt{Int}, \texttt{Text} \rangle\}\rangle\}\rangle\}$:

$$\begin{aligned} r = \{\langle \texttt{"Jones"}, \{ &\langle \texttt{"math"}, \{ \langle 1977, \texttt{"Russel"} \rangle, \\ &\qquad \langle 1978, \texttt{"Russel"} \rangle, \\ &\qquad \langle 1978, \texttt{"Doolittle"} \rangle\}\rangle, \\ &\langle \texttt{"science"}, \{\} \rangle, \\ &\langle \texttt{"physics"}, \{ \langle 1977, \texttt{"Martin"} \rangle, \\ &\qquad \langle 1978, \texttt{"Anderson"} \rangle\}\rangle\}\rangle, \\ \langle \texttt{"Smith"}, \{\langle \texttt{"physics"}, \{\} \rangle\}\rangle\} & \end{aligned}$$

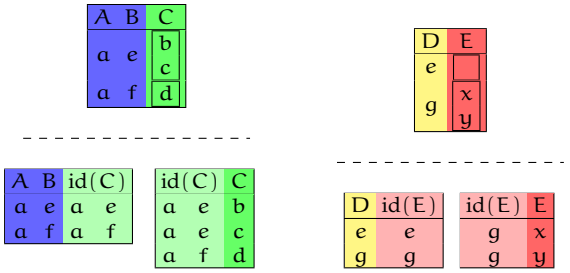
All sets in r are in PNF. Note that some nested sets are empty. Hulin notes that unnesting a nested relation into a single flat relation by repeatedly applying the nested relational unnest operator μ looses information about the nesting structure. Empty inner relations are not preserved. Instead, r is encoded by three flat sets (Figure 10).

The nested set is vertically sliced and each flat set represents one level of nesting. Since r is in PNF, the scalar key attributes of each set are used to link nesting levels. Note that keys of all enclosing levels are necessary to uniquely assign elements of r_3. Empty inner sets are represented by *absence*: the key $\langle \texttt{"Jones"}, \texttt{"science"} \rangle$ is present in r_2 but r_3 has no corresponding entries. Hulin shows that nested relations in PNF can be restored from this flattened representation without information loss.

2.1.2 *Query Flattening*

Query flattening has found most use in database theory. In this field, query flattening is used to investigate properties of nested relational query languages. Query flattening itself typically shows up as a byproduct that is not considered under practical aspects.

Paredaens and Van Gucht [PV92] show that any flat-to-flat query in nested relational algebra can be translated into one equivalent query in flat relational algebra. Any intermediate nested results used in the original nested query can be flattened. This means that nested relational algebra restricted to flat-to-flat queries is no more expressive than flat relational algebra. We say that nested relational algebra on flat-to-flat queries is a *conservative ex-*



(a) Nested relation R. (b) Nested relation S.

Figure 11: Flat encoding of nested relations in [Van99].

tension of flat relational algebra. Paredaens' and Van Gucht's translation approach is rather indirect and limited to flat-to-flat queries. As such, it is not a general solution to query flattening.

As a side note, we point out that Wong [Won96] shows a more general conservativity result for a nested relational calculus which also extends to queries over lists and bags. Cooper [Coo09] extends the conservativity result to flat-to-flat queries in a recursive higher-order language.

VAN DEN BUSSCHE Van Den Bussche [Van99] obtains the same conservativity result with a technique that is more relevant for our purposes. He shows that a nested-to-nested query in nested relational algebra can be simulated with multiple queries in relational algebra. The nested algebra consists of the core relational operators (π, σ, ρ, $\times$, $\cup$, $-$) extended with the bare minimum to work with nested relations: the operators ν (*nest*) and μ (*unnest*). The conservativity result for flat-to-flat queries follows as a byproduct of the simulation. We review the simulation briefly.

Consider the nested relations R and S and their flat encoding in Figure 11. Note that both relations are in PNF. Their flat encoding essentially corresponds to the one discussed in Section 2.1.1. Relation R with one complex attribute C is mapped two to relations. Non-complex attributes A and B are copied as unique identifiers for the complex attribute C, denoted as id(C). The inner relation uses these identifiers to relate the elements of the inner relations.

Van Den Bussche simulates a query in nested relational algebra with a bundle of flat queries expressed in a mixture of relational calculus and relational algebra. One flat query generates the flat representation of the result relation. For each complex attribute in the result, an additional flat query computes the corresponding inner relation. The result of the cartesian product $R \times S$ in Figure 12 is therefore encoded by three flat queries: one query for the top-level relation, and one query each for the complex attributes C and E. Note that the flat simulation of the cartesian product merely consists of the cartesian product of the outer relations while the representations of complex attributes are preserved unchanged. As a consequence, the indexes (*e.g.* id(C)) are no longer unique in the result. Data for complex attributes in inner relations is *shared*.

We consider a selection on atomic attributes of a nested relation in Figure 13. Here, the selection on the nested relation is translated into a selection on the outer flat relation while the inner relations are again unchanged. As a consequence, the inner relations contain stale tuples that are not referenced by the outer relation. While stale tuples are not an issue for correctness,

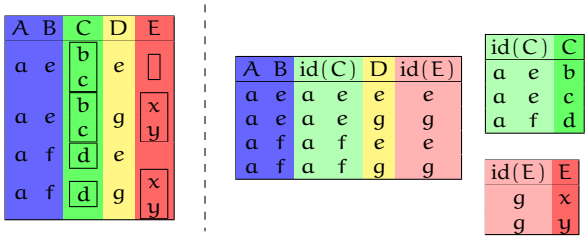


Figure 12: Product: $\nu_C(R) \times S$

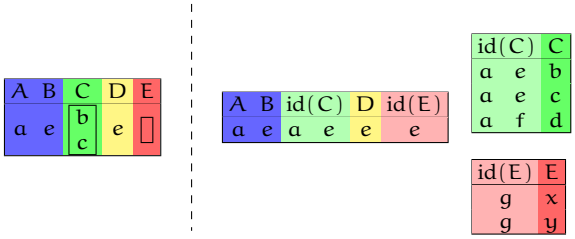


Figure 13: Selection: $\sigma_{B=D}(\nu_C(R) \times S)$

they are an issue for a practical implementation. It is easy to construct an example in which the flat encoding of a nested query result returned by the database is much larger than necessary. A query flattening implementation has to fetch these stale tuples from the database just to discard them while reconstructing the nested result.

Although Van Den Bussche shows that it is possible to simulate nested queries on a flat relational system *in principle*, his approach is not useful for *practical* query languages (Chapter 1). The simulation relies on set semantics and it is not clear how to extend it to lists or bags. Indeed, Cheney *et al.* [CLW14b, *Appendix A*] demonstrate that the simulation does not work for multisets. The nested algebra used is not a practical query language as it lacks aggregation, for example. Most significantly, the algebra is not compositional in the operator arguments of π and σ — relational operators can not be nested. This makes it impossible to directly express arbitrary *nested iteration* over nested collections. Nesting of queries has to be simulated by unnesting and re-nesting with μ and ν.

SUCIU Suciu [Suc97] proves a conservativity result for a nested relational calculus extended with recursion in the form of a *bounded fixpoint* operator. His approach is similar to Van Den Bussche's: Suciu describes an encoding of complex objects in flat relations. He then shows that any query in the nested relational calculus can be simulated by flat relational queries. Two things are particularly interesting about Suciu's work. First, besides its use as a proof technique, he proposes query flattening as an *implementation technique* for complex-object models on relational query engines. Second, in contrast to Van Den Bussche's algebra, Suciu's calculus is orthogonal and allows arbitrary nesting not only of data but also of iteration (*i.e.* nested data parallelism). We briefly review Suciu's query flattening approach.

Similar to approaches discussed earlier in this chapter, Suciu uses a flat encoding of nested collections based on *indexes*. This is in contrast to earlier work by Suciu [ST94; Suc95; Suc96] that uses a length-based encoding. Suciu defines a flat encoding for complex-object types (arbitrary combinations of base, tuple and set types). Base values of type s are encoded as singleton sets

TYPE	ENCODING TYPE	REPRESENTATION TYPE
`Int`	`{Int}`	`{Int}`
`{Int}`	$[\mathtt{I} \Rightarrow \mathtt{\{Int\}}]$	$\langle \mathtt{\{I\}}, \mathtt{\{\langle I, Int\rangle\}}\rangle$
$\mathtt{\{\langle Int, Int\rangle\}}$	$[\mathtt{I} \Rightarrow \langle \mathtt{\{Int\}}, \mathtt{\{Int\}}\rangle]$	$\langle \mathtt{\{I\}}, \langle \mathtt{\{\langle I, Int\rangle\}}, \mathtt{\{\langle I, Int\rangle\}}\rangle\rangle$
`{{Int}}`	$[\mathtt{I} \Rightarrow ([\mathtt{I} \Rightarrow \mathtt{\{Int\}}])]$	$\langle \mathtt{\{I\}}, \langle \mathtt{\{\langle I, I\rangle\}}, \mathtt{\{\langle I, I, Int\rangle\}}\rangle\rangle$

Table 4: Flat relational representation of complex-object types in [Suc97].

TYPE	VALUE	REPRESENTATION
`Int`	`42`	`{42}`
`{Int}`	`{10, 20, 30}`	$\mathtt{\{}i_1\mathtt{, }i_2\mathtt{, }i_3\mathtt{\}}$
		$\mathtt{\{}\langle i_1\mathtt{, 10}\rangle\mathtt{, }\langle i_2\mathtt{, 20}\rangle\mathtt{, }\langle i_3\mathtt{, 30}\rangle\mathtt{\}}$
$\mathtt{\{\langle Int, Int\rangle\}}$	$\mathtt{\{}\langle \mathtt{1, 10}\rangle\mathtt{, }\langle \mathtt{2, 20}\rangle\mathtt{, }\langle \mathtt{3, 30}\rangle\mathtt{\}}$	$\mathtt{\{}i_1\mathtt{, }i_2\mathtt{, }i_3\mathtt{\}}$
		$\mathtt{\{}\langle i_1\mathtt{, 1}\rangle\mathtt{, }\langle i_2\mathtt{, 2}\rangle\mathtt{, }\langle i_3\mathtt{, 3}\rangle\mathtt{\}}$
		$\mathtt{\{}\langle i_1\mathtt{, 10}\rangle\mathtt{, }\langle i_2\mathtt{, 20}\rangle\mathtt{, }\langle i_3\mathtt{, 30}\rangle\mathtt{\}}$
`{{Int}}`	`{{10, 20}, {}, {30}}`	$\mathtt{\{}i_1\mathtt{, }i_2\mathtt{, }i_3\mathtt{\}}$
		$\mathtt{\{}\langle i_1\mathtt{, }i_{11}\rangle\mathtt{, }\langle i_1\mathtt{, }i_{12}\rangle\mathtt{, }\langle i_3\mathtt{, }i_{31}\rangle\mathtt{\}}$
		$\mathtt{\{}\langle i_{11}\mathtt{, 10}\rangle\mathtt{, }\langle i_{12}\mathtt{, 20}\rangle\mathtt{, }\langle i_{31}\mathtt{, 30}\rangle\mathtt{\}}$

Table 5: Flat relational representation of complex-object values in [Suc97]

of type `{s}`. To encode sets, set elements are identified by unique indexes of some type I. The encoding of a set of type `{t}` can then be understood as a *finite* function $[\mathtt{I} \to \mathtt{t}]$ that maps an index to the encoding of a set element. Actually, a set of type `{t}` is encoded as a *partial* finite function $[\mathtt{I} \Rightarrow \mathtt{t}]$ that explicitly denotes its *domain* of type `{I}`. A partial function $[\mathtt{I} \Rightarrow \mathtt{\{s\}}]$ is represented as a pair of a flat relation `{I}` (the domain) and a function $[\mathtt{I} \to \mathtt{\{s\}}]$. The explicit representation of the function domain is necessary to preserve empty inner sets (Section 2.1.1).

In Table 4 we show examples of encoding and representation of various complex object types. The encoding of corresponding values is shown in Table 5. Note that a set of tuples $\mathtt{\{\langle Int, Int\rangle\}}$ is vertically fragmented: data for the individual tuple components is stored in individual binary relations. Effectively, the encoding implements a decomposed storage model [CK85]. Furthermore, we note that Suciu's encoding of a set of sets adds one layer of indirection compared to the encoding of Van Den Bussche that is not strictly necessary.

In Suciu's work, query flattening proceeds in two stages. Queries in the nested calculus are first translated into a variable-free nested relational algebra. Subsequently, this nested algebra is translated into a flat relational algebra on the indexed flat representation. The nested relational algebra is an algebra of functions in point-free form similar to the work of Buneman *et al.* [Bun+95]. Each sub-expression in a calculus term is translated into a function of its free variables. Nested iteration is translated into a higher-order combinator that maps its functional argument over a set and *replicates* any variables that are free in the iterator. This nested algebra is in turn translated into a flat algebra in points-free style. Central to the translation is the *Map Lemma* [Suc97, Lemma 5.4]: it describes how to translate operators that are iteratively applied to a set of operands to flat operations on the flat, indexed

encoding. For example, a select operator $map(\sigma)$ iteratively applied to a set of sets of type `{{t}}` translates into a selection on the flat set that encodes the inner set contents.

Suciu's translation is more interesting for our purposes than that of Van den Bussche: he uses an orthogonal calculus that expresses nested parallelism. Unfortunately, Suciu only sketches the translation and leaves out details.

MORE THEORETICAL QUERY FLATTENING Levy and Suciu [LS97] use the same query flattening approach in subsequent work to investigate containment properties of nested relational queries. Koch *et al.* [KLT16] have used query flattening to define *Incremental View Maintenance* for a nested relational calculus. They first flatten nested calculus queries and then define delta generation for the individual flat queries.

To summarize, we have reviewed two use cases for query flattening in a theoretical setting. First, query flattening is used as a tool to investigate relations between nested and flat queries by mapping one onto the other [Van99; Suc97]. Second, query flattening is used to simplify a problem on nested queries (containment, incremental view maintenance) by mapping it onto a simpler domain, namely flat queries [LS97; KLT16]. However, none of the approaches discussed so far has been developed under consideration of practical implementation aspects. We are not aware of any implementations of the approaches discussed here. In the following section we discuss approaches that are more fit for an actual implementation.

2.2 PRACTICAL APPROACHES

In the work discussed so far, query flattening merely played an auxiliary role. In this section, we discuss work that develops query flattening under practical considerations: as a tool to actually evaluate nested queries on flat relational systems. In particular, this requires to consider multisets instead of sets. We will discuss two approaches in more detail, as they have influenced our work considerably: *Query Shredding* in Section 2.2.1 and *Loop-Lifting* in Section 2.2.2.

2.2.1 *Query Shredding*

Cheney *et al.* [CLW14a] start with a higher-order nested relational calculus λ_{NRC} as a basis for language-integrated query systems. Contrary to approaches discussed in Section 2.1, the data model of λ_{NRC} is not based on sets but multisets to match real-world relational query languages (*e.g.* SQL). We adopt the syntax of our own language $\mathcal{CL}$ for λ_{NRC} examples. However, we stress that all collection operations work on multisets, not lists. To avoid confusion with list comprehensions, we write bag comprehensions as $\{\!|\ e \mid qs\ |\!\}$.

Cheney *et al.* define query flattening under the name *Query Shredding* for flat-to-nested λ_{NRC} queries. Prior to flattening, λ_{NRC} queries are normalized into a first-order subset of the calculus. In normal form, terms consist only of bag comprehensions, bag union ($\uplus$), emptiness tests `empty` and scalar operators. Normalized comprehensions have the form $\{\!|\ e \mid x \leftarrow t, p\ |\!\}_\alpha$ limited to one generator from a base table t and a single predicate. Each comprehension has a unique label. We let α, β range over comprehension labels. *Query Shredding* crucially relies on this normal form.

We first illustrate the basic idea of query shredding with a simple example. Consider the following normalized query q that returns a value of type $\{\!\{\langle \texttt{dept:Text}\rangle, \texttt{emps:}\{\!\{\langle \texttt{name:Text}, \texttt{sal:Int}\rangle\}\!\}\}\!\}$.

$$
\begin{aligned}
q = \{\!\{\ & \langle \texttt{dept} = \texttt{d.name}, \texttt{emps} = \{\!\{\ \langle \texttt{name} = \texttt{e.name}, \texttt{sal} = \texttt{e.sal}\rangle \\
& \qquad\qquad\qquad\qquad \mid \texttt{e} \leftarrow \texttt{table(employees)} \\
& \qquad\qquad\qquad\qquad , \texttt{e.sal} > 50000 \\
& \qquad\qquad\qquad\qquad , \texttt{e.dept} = \texttt{d.id}\ \}\!\}_{\beta}\rangle \\
& \quad \mid \texttt{d} \leftarrow \texttt{table(departments)}\ \}\!\}_{\alpha}
\end{aligned}
$$

Query results have the following form:

$$
\begin{aligned}
\{\!\{ & \langle \texttt{dept} = \texttt{"Sales"}, \texttt{emps} = \{\!\{ \langle \texttt{name} = \texttt{"Bob"}, \texttt{sal} = 60000\rangle, \\
& \qquad\qquad\qquad\qquad \langle \texttt{name} = \texttt{"Alice"}, \texttt{sal} = 80000\rangle\ \}\!\}\rangle, \\
& \langle \texttt{dept} = \texttt{"R\&D"}, \texttt{emps} = \{\!\{ \langle \texttt{name} = \texttt{"Lucy"}, \texttt{sal} = 100000\rangle, \\
& \qquad\qquad\qquad\qquad \langle \texttt{name} = \texttt{"John"}, \texttt{sal} = 120000\rangle\ \}\!\}\rangle, \\
& \langle \texttt{dept} = \texttt{"Acct"}, \texttt{emps} = \{\!\{\}\!\}\rangle\ \}\!\}
\end{aligned}
$$

Similar to work discussed in Section 2.1, Cheney *et al.* flatten nested queries into a bundle of flat queries. In contrast to earlier work, though, *Query Shredding* does not translate into relational algebra but into a flat multiset calculus. For each multiset type constructor in the result type of the query, *Query Shredding* produces one flat calculus query. *Indexes* link the individual flat multisets computed by those queries and allow the reconstruction of nested results.

Shredding query q results in two flat queries. The *outer* query q_1 returns the departments and replaces the nested comprehension with a scalar index $I_\alpha(v)$. For the comprehension labeled α, $I_\alpha(v)$ denotes an index value constructed from scalar value v.

$$
\begin{aligned}
q_1 = \{\!\{\ & \langle \texttt{dept} = \texttt{dept} = \texttt{d.name}, \texttt{emps} = I_\alpha(\texttt{d})\rangle \\
& \mid \texttt{d} \leftarrow \texttt{table(departments)}\ \}\!\}
\end{aligned}
$$

The *inner* flat query q_2 computes the employees of every department. It returns pairs of index values and the actual employee data. Note that q_2 includes the generator of the outer comprehension in q. Each element of the result of q_2 is paired with an index value $I_\alpha(\texttt{d})$ derived from the department. Both queries q_1 and q_2 share the same index value for each individual department.

$$
\begin{aligned}
q_2 = \{\!\{\ & \langle I_\alpha(\texttt{d}), \langle \texttt{name} = \texttt{e.name}, \texttt{sal} = \texttt{e.sal}\rangle\rangle \\
& \mid \texttt{d} \leftarrow \texttt{table(departments)} \\
& , \texttt{e} \leftarrow \texttt{table(employees)} \\
& , \texttt{e.sal} > 50000 \\
& , \texttt{e.dept} = \texttt{d.id}\ \}\!\}
\end{aligned}
$$

Flat queries q_1 and q_2 produce the following flat multisets, with i_1, i_2, i_3 denoting index values derived from departments. Nested results are reconstructed by joining both bags on the indexes. Note that the empty multiset for "Acct" is encoded by the absence of index i_3 in the inner result.

$$\{\!|\langle \texttt{dept} = \texttt{"Acct"}, \texttt{emps} = i_1\rangle, \langle \texttt{dept} = \texttt{"R\&D"}, \texttt{emps} = i_2\rangle, \langle \texttt{dept} = \texttt{"Acct"}, \texttt{emps} = i_3\rangle|\!\}$$

$$\{\!|\langle i_1, \langle \texttt{name} = \texttt{"Bob"}, \texttt{sal} = 60000\rangle\rangle, \langle i_2, \langle \texttt{name} = \texttt{"Lucy"}, \texttt{sal} = 100000\rangle\rangle, \langle i_1, \langle \texttt{name} = \texttt{"Alice"}, \texttt{sal} = 80000\rangle\rangle, \langle i_2, \langle \texttt{name} = \texttt{"John"}, \texttt{sal} = 120000\rangle\rangle|\!\}$$

Cheney *et al.* organize the flat queries in a *shredded package*: the original type of the nested query in which each multiset type constructor is annotated with the corresponding flat query:

$$\{\!|\langle \texttt{dept:Text}, \texttt{emps:}\{\!|\langle \texttt{name:Text}, \texttt{sal:Int}\rangle|\!\}^{q_2}\rangle|\!\}^{q_1}$$

In normalized λ_{NRC} terms, the term structure reflects the type structure of the result as seen in q. Each multiset type constructor corresponds to one comprehension (or a union of comprehensions) in the term. *Query Shredding* exploits this correspondence and splits the original nested query term at each multiset type constructor. The comprehension corresponding to that multiset type constructor is split into an outer query and inner queries as seen in q_1 and q_2: in the outer query, any nested comprehensions are replaced with an index value derived from the generator of the comprehension. Inner queries derive the same index value to correlate outer and inner queries. This establishes the index relationship between the individual flat results.

Shredding a nested comprehension includes *all* enclosing generators in index values. Consider shredding the following query at level $n-1$.

$$\begin{aligned}&\{\!|\ \ldots\ \{\!|\ \{\!|\ e \mid x_n \leftarrow t_n, p_n\ |\!\}_{\alpha_n}\\ &\qquad \mid x_{n-1} \leftarrow t_{n-1}, p_{n-1}\ |\!\}_{\alpha_{n-1}} \cdots\\ &\mid x_1 \leftarrow t_1, p_1\ |\!\}_{\alpha_1}\end{aligned}$$

Each evaluation of the head expression $\{\!|\ e \mid x_n \leftarrow t_n, p_n\ |\!\}$ is determined by the bindings $x_1, \ldots, x_{n-1}$ of all enclosing generators. Shredding at level $n-1$ derives an inner query that computes elements for all inner lists. To uniquely identify those inner lists, indexes at level $n-1$ are obtained from bindings of all enclosing generators. Hence, shredding obtains the following outer and inner query at level $n-1$:

$$\begin{aligned}q_{n-1} = {}& \{\!|\ \langle I_{\alpha_{n-2}}(x_1, \ldots, x_{n-2}), I_{\alpha_{n-1}}(x_1, \ldots, x_{n-1})\rangle\\ &\mid x_1 \leftarrow t_1, p_1, \cdots, x_{n-1} \leftarrow t_{n-1}, p_{n-1}\ |\!\}\\ q_n = {}& \{\!|\ \langle I_{\alpha_{n-1}}(x_1, \ldots, x_{n-1}), e'\rangle\\ &\mid x_1 \leftarrow t_1, p_1, \cdots, x_{n-1} \leftarrow t_{n-1}, p_{n-1}, x_n \leftarrow t_n, p_n\ |\!\}\end{aligned}$$

In *Query Shredding*, comprehension labels guarantee unique index values. Consider the following query involving a bag union. Note that generators in both outer comprehensions use the same base table t_1.

$$\{\!|\ \{\!|\ e_2 \mid y \leftarrow t_3\ |\!\}_{\alpha_2} \mid x \leftarrow t_1\ |\!\}_{\alpha_1} \uplus \{\!|\ \{\!|\ e_4 \mid z \leftarrow t_5\ |\!\}_{\beta_2} \mid x \leftarrow t_1\ |\!\}_{\beta_1}$$

The following outer query is obtained by shredding:

$$\{\!|\ I_{\alpha_1}(x) \mid x \leftarrow t_1\ |\!\} \uplus \{\!|\ I_{\beta_1}(x) \mid x \leftarrow t_1\ |\!\}$$

If indexing were to depend only on the generator binding x, indexes for the outer comprehensions would overlap. Including the comprehension label in the index ensures unique indexes without overlap.

Query Shredding maps flat multiset queries to SQL queries with a simple scheme: flat comprehensions obtained via shredding are isomorphic to SQL. Translation to SQL is merely a syntactical mapping. Index values are derived from keys of base tables. *Query Shredding* synthesizes uniform integer index values by enumerating the underlying combinations of base table keys with SQL's `row_number()` window function. Cheney *et al.* also consider *natural* indexes which are tuples of base table keys but do not explore this option further. Hence, all SQL queries obtained by shredding nested comprehensions include window functions.

2.2.1.1 *A Critique of Query Shredding*

We find *Query Shredding* to be rather simple and elegant. The structure of flat queries obtained via shredding is directly related to the original (normalized) nested query and easy to predict. After shredding, flat comprehensions translate to relatively simple SQL queries. Apart from window functions that compute index values, *Query Shredding* produces regular benign join queries in SQL. In addition, Cheney *et al.* give a rigorous proof of correctness for query shredding, showing that the shredding translation preserves the semantics of the original nested queries.

The language λ_{NRC} that can be handled by *Query Shredding*, however, falls short of being a satisfying query language. Its limitation to bag semantics and lack of grouping, duplicate elimination, aggregation and other functionality makes it impossible to express complex queries. Cheney *et al.* assume that *Query Shredding* can be extended towards grouping, aggregation and list semantics. However, we are not aware of any follow-up work in this direction.

Query Shredding fundamentally relies on the correspondence of terms and types after normalization. It is not clear whether this property can be extended to a richer query language. Prior work on normalization of flat-to-flat queries by Cooper [Coo09] and Cheney *et al.* [CLW13] is based on similarly restricted query languages. Cheney *et al.* [CLW13] prove normalization for flat-to-flat queries to be complete and correct for a restricted query language. However, they do not uphold those guarantees if the language is extended with grouping, sorting and aggregation. Hence, we believe that *Query Shredding* can not handle a more expressive query language without fundamental changes to the approach.

2.2.2 *Loop-Lifting*

Among practical approaches to query flattening, *Loop-Lifting* takes a rather prominent place in the literature. In this section, we first review literature on *Loop-Lifting*. We then go on to explain the core concepts of *Loop-Lifting* in some detail to provide a basis for a comparison with our work.

2.2.2.1 *Literature Survey on* Loop-Lifting

Grust *et al.* [GST04] originally describes *Loop-Lifting* as a compilation technique that translates XQuery into relational queries. While preceding work focused on relational encodings of XML documents and the evaluation of XPath axes [Gru02], *Loop-Lifting* focuses on the iterative core of XQuery: FLWOR comprehensions express *potentially nested* iteration over ordered sequences without side effects. *Loop-Lifting* is centered around a relational encoding of iteration that transforms iterative into set-oriented evaluation.

From the get-go, *Loop-Lifting* supports an extensive subset of XQuery, including aggregation, order-aware operations, set operations and duplicate elimination next to XML-specific constructs.

Grust *et al.* [GST04] provide an account of *Loop-Lifting* that maps XQuery constructs directly to SQL expressions. Grust [Gru05] replaces SQL with a relational algebra on multisets as a backend-independent intermediate language. From this algebra, *Loop-Lifting*'s implementation Pathfinder offers two paths for actual backend code generation. First, Pathfinder has been integrated with MonetDB [MKB09] to form the XQuery processor MonetDB/XQuery [Bon+06]. Here, Pathfinder directly emits MonetDB's MAL algebra and exploits XML-specific operators in the MonetDB kernel (*e.g. staircase join* [GVT03]). Second, a SQL code generator [Gru+07] allows to execute *Loop-Lifting* queries on a SQL:2003 database system. Both approaches have been found to provide efficient and scalable XQuery implementations [Rit11].

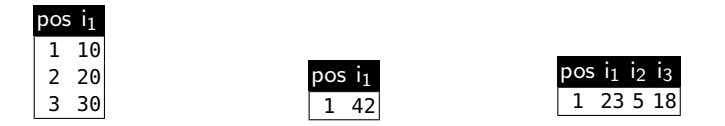
Originally, *Loop-Lifting* is tailored to XQuery and the peculiarities of its data model: sequences are flat, have only scalar elements and singleton sequences and scalar values are equivalent. *Loop-Lifting* has since been extended to other query languages in the Ferry project [Gru+09]. Most importantly, the data model considered by *Loop-Lifting* has been extended from flat XQuery sequences to arbitrarily nested lists. *Loop-Lifting* supports an orthogonal query language with sorting, grouping, aggregation and duplicate elimination. Furthermore, Rittinger [Rit11] describes a relational encoding of *closures* and Ulrich [Ulr11] adds *sum-of-product* types. In the presence of nested query results, *Loop-Lifting* generates a bundle of flat queries.

A number of experimental language-integrated query systems make use of *Loop-Lifting*. Grust *et al.* [GRS10] describe an implementation of Microsoft LINQ that integrates nested relational queries with queries over XML documents. Ulrich [Ulr11] extends the query facility in Links to nested results, grouping, aggregation and list semantics using *Loop-Lifting*. Giorgidze *et al.* [Gio+11a] describe *Database-Supported Haskell* (DSH), a type-safe query facility for Haskell based on a deep embedding of list combinators and comprehensions. Their version of DSH is a predecessor of DSH as discussed in Chapter 1. An alternative implementation of Ruby's ActiveRecord query facility is described by Grust and Mayr [GM12; May13]. *Loop-Lifting* has found use beyond XQuery compilation and language-integrated query systems: Grust and Rittinger [GR11] propose *observational debugging* for SQL queries based on *Loop-Lifting*.

2.2.2.2 *Algebraic Iteration*

Loop-Lifting is a *compositional* compilation scheme that offers an isolated compilation rule for each syntactic category of an orthogonal query language. In contrast to *Query Shredding*, *Loop-Lifting* does not require the query to be in a normal form. We review the core concepts of *Loop-Lifting* — its algebraic handling of nested iteration and nested data — based on simple examples. We refer to Rittinger [Rit11] for a more comprehensive account of *Loop-Lifting*.

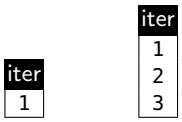
We start with a relational encoding of flat lists and scalar values as depicted in Figure 14. A list element is represented by exactly one relational tuple. List elements are annotated with an explicit encoding of list order. Individual fields of (nested) records are mapped to relational attributes. Base tables are mapped to the list encoding by enumerating their elements in the order of the table's primary key and using that enumeration as posi-



`[10,20,30]` `42` $\langle$`23,`$\langle$`5,18`$\rangle\rangle$

(a) `[Int]` (b) `Int` (c) $\langle$`Int,`$\langle$`Int,Int`$\rangle\rangle$

Figure 14: Relational encoding of ordered lists and scalar types in *Loop-Lifting*.



(a) s_0 (b) s_1

Figure 15: *Loop-Lifting*: Relational encoding (*loop* relations) of iteration scopes s_0, s_1.

tions. *Loop-Lifting* provides an algebraic, set-oriented account of iteration over lists. Consider the following expression:

$$\overbrace{\texttt{[}\underbrace{\texttt{x+42}}_{s_1}\texttt{ | x} \leftarrow \texttt{[10,20,30]]}}^{s_0}$$

Loop-Lifting considers every expression to be evaluated in an *iteration scope*. The comprehension, including its generator expression `[10,20,30]`, is evaluated once in the outer iteration scope s_0. Scope s_0 encompasses a single dummy iteration. Expression `x+42`, on the other hand, is evaluated independently three times under different bindings for `x` in the inner iteration scope s_1. *Loop-Lifting* enumerates iterations in an iteration scope and represents them as a single-column *loop* relation (Figure 15).

For each (sub)-expression e, a syntax-directed compilation rule derives a relational algebra plan fragment that encodes the evaluation of e in every iteration of the current iteration scope. We depict the plan fragments for individual sub-expressions in Figure 16. Plan fragment q_1 establishes the representation of the generator expression `[10,20,30]` in the outer iteration scope s_0. The relational representation of variable `x` in scope s_1 is derived by the plan fragment q_2 (Figure 16b). The numbering operator $\#_{:\langle\rangle}$ computes an enumeration of the list elements in the generator expression, corresponding to an enumeration of iterations over this list. In the resulting table, every tuple represents one iteration of the loop over the list `[10,20,30]`. The plan fragment in Figure 16c derives the representation of sub-expression `42` in iteration scope s_1. The scalar value is *replicated* over the *loop* relation to provide a copy in each iteration.

Plan fragments q_2 and q_3 encode bindings for the variable `x` and a copy of the constant `42` for each iteration of s_1. Plan fragment q_4 aligns corresponding iterations and performs the addition. The addition operator is applied in a *set-oriented* manner by the relational projection operator π. Plan fragment q_5 maps the result of the individual iterations back to the enclosing iteration scope to obtain the encoding of the result list of three elements. It computes a mapping from iteration identifiers of the *inner* scope s_1 to list positions of the *outer* scope s_0 and joins this mapping with q_4. Note that this backmapping step also restores the order encoding of the generator list.

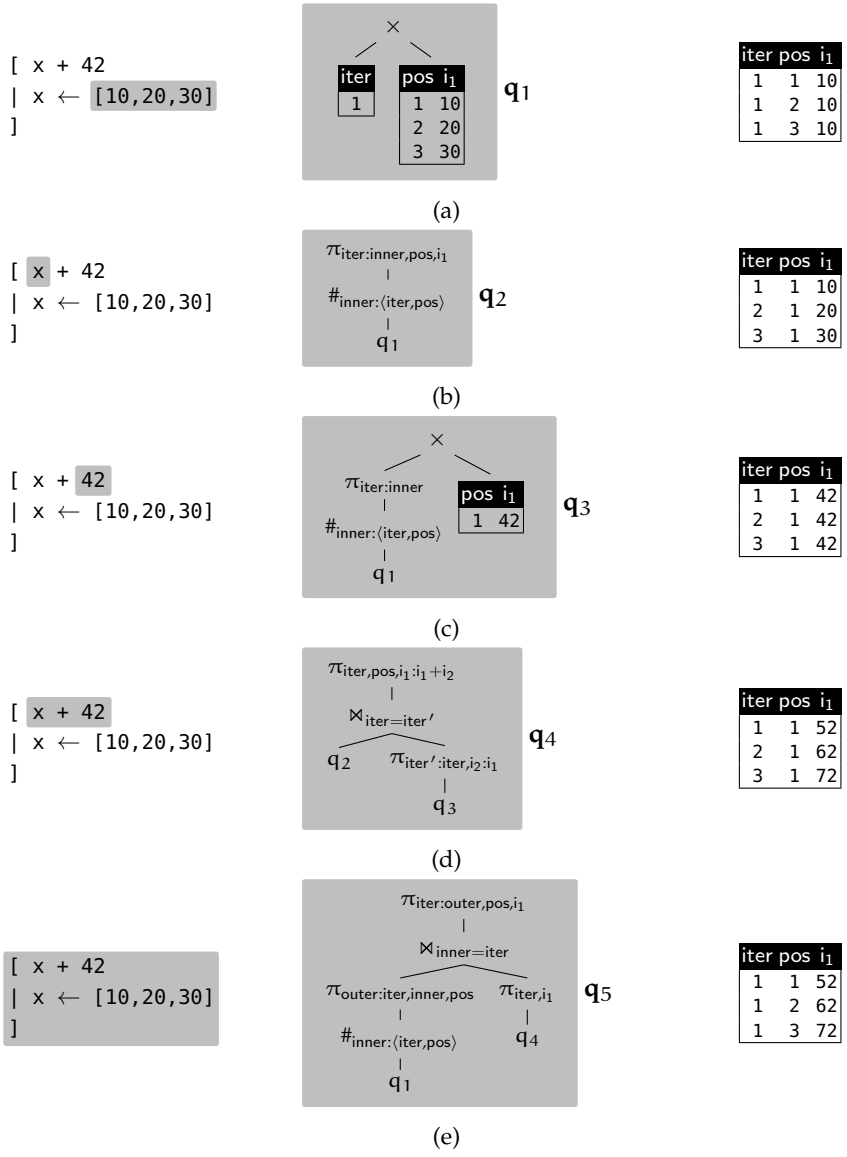


Figure 16: Translation of a flat comprehension by *Loop-Lifting*. The relational plan fragment is generated for the highlighted expression on the left and computes the relation on the right.

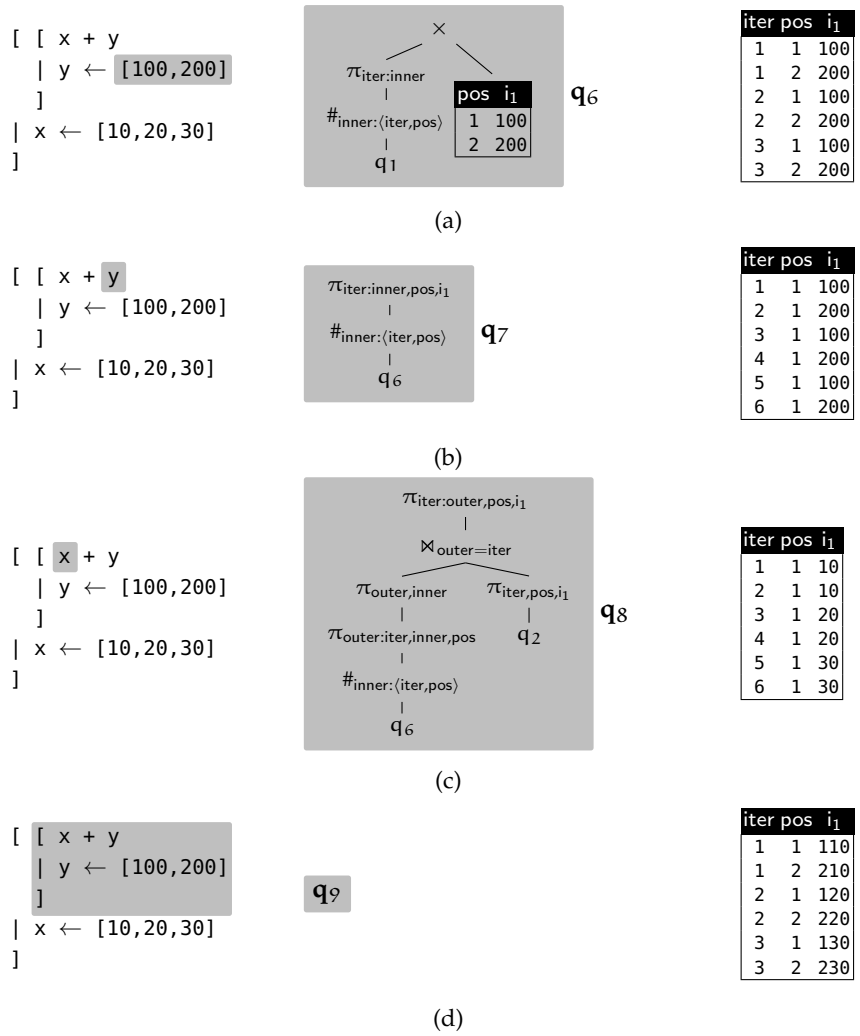


Figure 17: Translation of nested comprehensions by *Loop-Lifting*.

2.2.2.3 *Nested Iteration*

Consider the following example with nested comprehensions:

$$\overbrace{\texttt{[[x + y | y} \leftarrow \texttt{[100, 200]] | x} \leftarrow \texttt{[10, 20, 30]]}}^{s_0}$$
$$\underbrace{\texttt{[[x + y}}_{s_2} \texttt{ | y} \leftarrow \texttt{[100, 200]]}}_{s_1} \texttt{ | x} \leftarrow \texttt{[10, 20, 30]]}$$

The nested comprehensions define three iteration scopes s_0, s_1 and s_2. As for the previous example, we depict plan fragments and intermediate relations for sub-expressions in Figure 17. Note that plan fragments for the outer comprehension (list `[10, 20, 30]` and the *loop* relation for scope s_1) are the same as in Figure 16.

The constant expression `[100, 200]` is compiled in fragment q_6 just as the constant `42` before (Figure 17a): the relational encoding of the constant is *replicated* into an independent copy for each iteration in the current iteration scope s_1. From q_6, fragment q_7 derives the encoding of variable `y`: it enumerates all copies of `[100, 200]` to derive new iteration identifiers for scope s_2. The resulting relation encodes the binding for variable `y` for all six iterations of scope s_2.

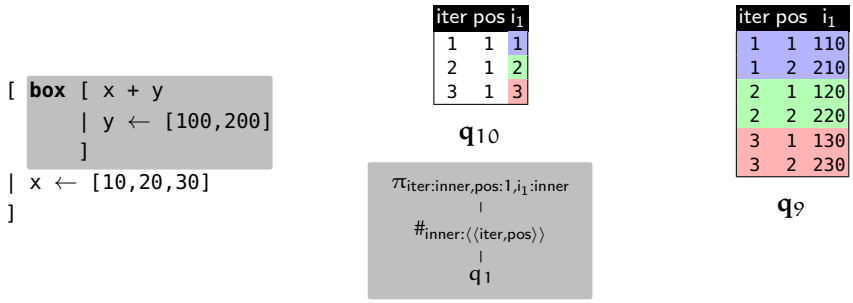


Figure 18: *Loop-Lifting*: Splitting plans to derive the flat representation of a nested list.

In iteration scope s_2, expression `x + y` is evaluated in an environment with bindings for the variables `x` and `y`. Considering only variable bindings by comprehensions, *lexical* and *iteration* scopes coincide. The encoding of variable `x` obtained in fragment q_2 (Figure 16b), however, is bound to iteration scope s_1. Plan fragment q_8 (Figure 17c) derives a relation mapping identifiers of the *outer* scope (s_1) to identifiers of the *inner* scope (s_2). A join with q_2 creates a copy of bindings for `x` in s_1 for each corresponding inner iteration in s_2. Variable bindings from the enclosing iteration scope are lifted into the new scope by replicating the binding. We refer to this as *environment lifting*.

Plan fragments q_7 and q_8 encode bindings for variables `y` and `x` for all iterations of s_2. As in our previous example, a join aligns both relations along their iteration identifiers to enable a set-oriented evaluation of the addition operator (not depicted). The same mapping relation derived in q_8 for environment lifting is utilized to map the result of the addition back to iteration scope s_1 (Figure 17d). We omit the explicit presentation of plan fragment q_9 and refer to the analogous mapping fragment in Figure 16e.

In summary, *Loop-Lifting* implements nested iteration by *replicating data*. Generator expressions (constant lists, base tables) are replicated as well as variable bindings. Concretely, replication here means the computation of cartesian products (Figure 17a) and joins (Figure 17b). *Loop-Lifting* heavily relies on numbering operators to compute iteration identifiers that map between iteration scopes and align iterations.

2.2.2.4 *Constructing Nested Lists*

The query in Figure 16 returns a flat list that is readily encoded by a single flat relation. In contrast, the result of query in Figure 17 is nested. To encode nested lists, *Loop-Lifting* splits the plan into multiple flat relations linked by indexes. Preceding algebraic compilation, *Loop-Lifting* identifies expressions e that require a split of the plan and inserts annotations **box** e (Figure 18)

Recall plan fragment q_9 (Figure 17d) that encodes the result of the nested comprehension. It encodes the content of the three inner lists produced by the inner comprehension. However, it does not yet encode the list of type `[[Int]]` of those inner lists: an element (*i.e.* each of the inner lists) occupies more than one relation tuple. The proper representation is derived by compiling the **box** annotation on q_9 (Figure 18). In plan fragment q_{10}, it derives an outer relation that defines the structure of the nested list. Outer relation (q_10) and inner relation (q_9) are linked by indexes derived from

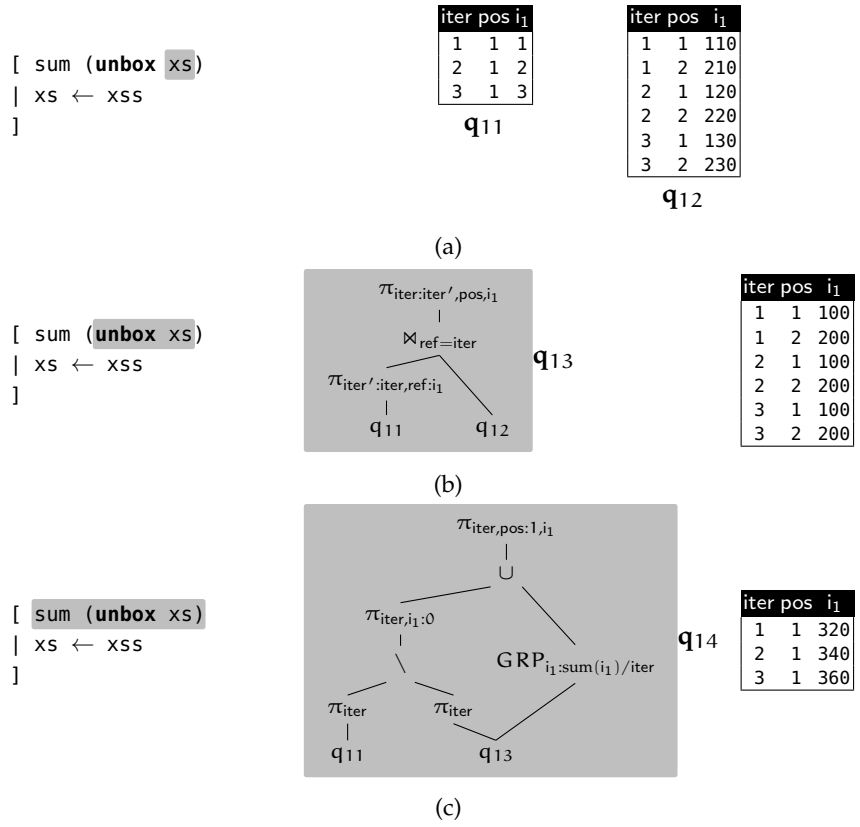


Figure 19: *Loop-Lifting*: Merging split plans.

iteration identifiers and computed by the numbering operator $\#_{:\langle\rangle}$ — *flat* or synthetic indexes in the terminology of Cheney *et al.* [CLW14a]. When translating queries on base tables, indexes are originally synthesized from base table keys. We point out that — contrary to a remark by Cheney *et al.* [CLW14b] — indexes in *Loop-Lifting* do not only link adjacent levels of nesting. Indexes are derived from iteration identifiers. As we have seen in the previous paragraph, iteration identifiers for nested iteration are obtained from iteration identifiers of the enclosing iteration scope and positions of the generator expression. Hence, *Loop-Lifting* indexes incorporate information from all levels of nesting.

To obtain the algebraic plan for the complete expression, plan fragment q_{10} is mapped to the outer scope s_0 while the inner fragment q_9 remains unchanged. In effect, *Loop-Lifting* generates a bundle of two relational algebra plans that encode the nested list.

QUERYING NESTED LISTS Consider the following expression which applies the `sum` aggregate iteratively to all inner lists `xs`:

$$\overbrace{\texttt{[}\ \underbrace{\texttt{sum xs}}_{s_1}\ \texttt{|}\ \texttt{xs} \leftarrow \texttt{xss}\ \texttt{]}}^{s_0}$$

Variable `xss : [[Int]]` denotes the result of the query in Figure 17:

```
xss ≡ [[110,210],[120,220],[130,230]]
```

To evaluate `sum`, the content of the inner lists is required. The two plans that encode this nested list are merged. *Loop-Lifting* annotates expressions e that require merging of plans as `unbox` e (Figure 19).

In Figure 19a, plan fragment q_{11} establishes the loop-lifted representation of variable `xs` as usual. Plan fragment q_{13} in Figure 19b dereferences the indexes in column i_1 of the outer relation q_{11} by joining with the inner relation q_{12}. Based on the unboxed representation, plan fragment q_{14} (Figure 19c) computes the aggregate `sum` independently for all iterations by grouping on the iteration identifiers `iter`. Note that the plan fragment accounts for empty inner lists (represented by the *absence* of iteration identifiers) and includes the default value `0` for empty lists.

2.2.2.5 *A Critique of* Loop-Lifting

Loop-Lifting supports expressive query languages like $\mathcal{CL}$. The translation can handle arbitrary expressions of the query language and does not depend on a normalization step. *Loop-Lifting* has been used successfully to implement relational XQuery processors. However, we argue that it suffers from conceptional problems that limit its theoretical appeal as well as its practical usability.

While *Query Shredding* pairs a simple query language with an equally simple flattening scheme, *Loop-Lifting* pairs a rich query language with a complex flattening scheme. Most striking about *Loop-Lifting* is its lack of abstraction. The source language is a rich orthogonal calculus with a complex data model (lists, records). Its target language is a simple relational algebra over unordered, multisets. *Loop-Lifting* bridges the considerable gap between these languages and data models in one large, monolithic step. No intermediate abstractions are used. In this monolithic compilation step, *Loop-Lifting* rules handle multiple aspects of query flattening at once.

- Nested records are mapped to flat relational schemas.
- Nested iteration is implemented algebraically by replicating data and lifting the environment.
- A flat representation of nested data based on synthetic keys is generated (**`box`**, **`unbox`**).
- List order is explicitly encoded and maintained using numbering operators.

Loop-Lifting's algebraic compilation rules that handle these tasks are complex and hard to comprehend. As an example, Figure 20 shows the translation rule for the iteration construct in LINQ. *Loop-Lifting*'s compilation rules are also hard to extend. For example, the synthetic index scheme based on numbering operators is fixed in the compilation rules. Due to the conceptual complexity embedded in this single translation step, it appears hard to reason about the rules.

Relational plans generated by *Loop-Lifting* feature a complex structure with large numbers of operators. The plan fragments we show in Figures 16 and 17 for very simple queries combine into query plans of considerable size. Indeed, Grust [Gru05] notes that flat plans for moderately complex queries consist of hundreds of operators and that the plan size has a negative impact on performance. Moreover, not just the size but also the *nature* of the plans is problematic: As we saw in Figure 17, *Loop-Lifting* plans fix a nested-loop evaluation strategy for nested iteration. In query engines,

$$\frac{\begin{array}{c}\{\ldots, x \mapsto (q_x, cs_x, ts_x), \ldots\}; loop \vdash e_1 \Rrightarrow (q_1, cs_1, ts_1) \\ q_v \equiv \varrho_{\mathsf{inner}:\langle \mathsf{iter},\mathsf{pos}\rangle}(q_1) \quad map \equiv \pi_{\mathsf{iter},\mathsf{inner}}(q_v) \quad loop_v \equiv \pi_{\mathsf{iter}:\mathsf{inner}}(map) \\ \Gamma_{lift} \equiv \{\ldots, x \mapsto (\pi_{\mathsf{iter}:\mathsf{inner},\mathsf{pos},cs_x}(q_x \bowtie_{\mathsf{iter}} map), cs_x, ts_x), \ldots\} \\ \Gamma_{lift} + \{v \mapsto (@_{\mathsf{pos}:1}(\pi_{\mathsf{iter}:\mathsf{inner},cs_1}(q_v)), cs_1, ts_1)\}; loop_v \vdash e_2 \Rrightarrow (q_2, cs_2, ts_2)\end{array}}{\begin{array}{c}\{\ldots, x \mapsto (q_x, cs_x, ts_x), \ldots\}; loop \vdash e_1\texttt{.Select(}v\texttt{ => }e_2\texttt{)} \Rrightarrow \\ (\pi_{\mathsf{iter},\mathsf{pos},cs_2}(\varrho_{\mathsf{pos}:\langle \mathsf{inner}\rangle/\mathsf{iter}}(\pi_{\mathsf{inner}:\mathsf{iter},cs_2}(q_2) \bowtie_{\mathsf{inner}} map)), cs_2, ts_2)\end{array}} \quad (7)$$

Figure 20: *Loop-Lifting* rule for LINQ's iteration construct, equivalent to `[ e2 | v ← e1 ]` (reproduced from [GRS10]).

nested-loop evaluation basically circumvents efficient access paths and join algorithms. In addition, plans are littered with expensive numbering operators used to derive iteration identifiers, indexes and list order. As noted by Grust *et al.* [GMR09], shape and size of plans basically overwhelm the optimizers of common database systems.

To transform *Loop-Lifting* plans into a form that is reasonably executable, a number of algebraic simplifications have been proposed [Gru05; GMR09]. Most comprehensively, Rittinger [Rit11] describes a set of heuristic rewrite rules based on inferred plan properties (*e.g.* functional dependencies, keys). Rewrite rules aim to decorrelate queries, reduce the effort for order maintenance and decrease the overall plan size. These rules are implemented in Pathfinder's optimizer.

For list queries with nested results, the results of Pathfinder's optimizations seems to be mixed. Cheney *et al.* [CLW14a] provide evidence that Pathfinder plan simplification is unable to eliminate numbering operators even for some simple nested queries. Numbering operators sit atop cartesian products and prevent merging selections into products to form joins. This forces the query engine to sort (*i.e. materialize*) intermediate results of quadratic size. Rittinger [Rit11] notes that numbering operators can't be eliminated for split plans because indexes need to be kept consistent across individual plans. Pathfinder's optimization scheme is based on a fixed pipeline of rewrites with complex dependencies between individual phases. As noted by Rittinger [Rit11], this causes the optimizer to miss opportunities for plan simplification. We believe that the optimization problems are caused mostly by the low-level nature of the algebraic plans that *Loop-Lifting* compiles into directly. The original macro query structure is blurred by index and order maintenance as well as compositionality artifacts.

2.2.3 *Dodo*

Van Ruth [Van06] describes *Dodo*, a further approach to query flattening. *Dodo* appears very similar to earlier work by Suciu [Suc97] (Section 2.1.2) but is phrased with the notions of category theory. *Dodo* is based on a flat encoding of complex objects in *binary* relations. Each binary relation encodes a partial finite function from indexes to atomic values. The notion of an encoding function is extended to *frames* that describe a mapping from an index to a complex value in its decomposed flat representation. Calculus queries on the complex object model are first transformed into a point-free nested algebra and subsequently into a flat point-free algebra on the flat representation. The former translation implements nested iteration by replicating bindings of free variables (*environment lifting*). The latter translation is described as in Suciu's approach as a composition of the functions of the point-free algebra with the functions of the flat representation.

Van Ruth defines a mapping of the binary relations to the binary data model of MonetDB that is similar to earlier work by Boncz *et al.* [BWK98]. Backend code generation is performed as a direct translation from the flat algebra on binary relations to MonetDB's *MIL* algebra [BK99]. As in Suciu's approach, the query flattening translation of *Dodo* fundamentally relies on binary relations for its flat representation. Considering actual query processing, this might be a good fit for the particular processing model of MonetDB. However we will argue in Section 4.3 that an early binary decomposition is not helpful when targetting a general relational query language like SQL.

We are not aware of a performance evaluation of *Dodo*. As noted by Van Ruth [Van06], query flattening in *Dodo* is similar to *Loop-Lifting*. We expect that relational plans produced by *Dodo* exhibit the same problematic characteristics as those produced by *Loop-Lifting*.

2.3 OUTLOOK

We believe that the normalization approach in *Query Shredding* is too limiting and that a translation that can handle arbitrary expressions in the input language is necessary. The three descriptions of compositional query flattening (Suciu [Suc97], *Loop-Lifting* and *Dodo*) we have reviewed in this chapter are structurally similar. Nested iteration is implemented with algebraic bulk operators by replicating base data and bindings for free variables. *Loop-Lifting* shortcuts the translation into a nested algebra to eliminate variables and tracks free variables explicitly in its monolithic translation.

All approaches have deficits, however, that limit their usability. In the remainder of this thesis, we try to address those deficits. In Chapter 3 we review the flattening transformation and sketch a translation scheme that makes query flattening more tractable. In following chapters we build on this foundation and address the optimization problem as well as the computation of indexes and maintenance of list order.

3 THE FLATTENING TRANSFORMATION

The problem we aim to solve is to support an expressive, nested query language with ordered collections on off-the-shelf relational query engines. In Chapter 2 we surveyed the literature on *query flattening* and found prior work to be lacking. In this chapter, we look at a program transformation from a different domain — *nested data parallelism* — that can be adopted for our purposes.

3.1 FLATTENING NESTED DATA PARALLELISM

Data parallelism is expressed by an *apply-to-each* construct that describes the side-effect free iteration over all elements of a collection. Consider the following expression in which some function `f` is applied to all elements of a collection `xs`.

$$[\ f\ x\ |\ x \leftarrow xs\ ]$$

If function `f` does not contain further iterators we say that the expression exhibits *flat data parallelism*. If `f` itself is parallel — that is, it contains iterators of the above form — we say that it exhibits *nested data parallelism*. A considerable number of algorithms naturally exhibit nested data parallelism [Ble90]. Nested data parallelism coincides with nested collections.

Nested data parallelism is challenging to implement. It needs to be mapped to parallel execution platforms — parallel hardware like GPUs, multi-threaded runtimes or distributed systems — that only provide flat parallelism. Nested data parallelism is *irregular*: in general, the amount of work performed by each application of `f` in the example above differs. It is hard to balance that work in order to fully exploit all parallel capabilities.

The *flattening transformation* is a static program transformation that eliminates nested data parallelism: programs with nested data parallelism are rewritten into an equivalent program that only uses flat data parallelism. The example above is rewritten into expression `f' xs` where `f'` is a function that applies the computation of `f` to all elements of its argument in parallel and uses only flat parallelism. The flattening of parallel computation coincides with the flattening of collections: as part of the transformation, nested collections are mapped to a flat representation.

The flattening transformation has been described for a variety of languages: Blelloch's data-parallel language *NESL* [Ble+94] and a subset of Haskell [Pey+08], for example. In all descriptions of the flattening transformation, the input languages share the same core: central is an *apply-to-each* construct that expresses data parallelism, a library of collection operations and a data model centered around ordered, nested collections (vectors, arrays, sequences). This is the same class of *collection-oriented* languages described earlier in the context of collection programming Chapter 1.

Flattening has been first described by Blelloch and Sabot [BS89], albeit only informally. To the best of our knowledge, Prins and Palmer [PP93] are the first to give a formal account of the flattening transformation as a set of rewrite rules. Subsequent work has extended the flattening transformation

to richer data models including recursive types [CK00] and higher-order functions [Les05].

3.2 THE FLATTENING TRANSFORMATION BY EXAMPLE

Our query language $\mathcal{CL}$ consists of comprehensions that describe nested iteration as well as number of scalar and list primitives. It perfectly matches the class of languages that can be handled by the flattening transformation. If anything, our job is easier because $\mathcal{CL}$ has no user-defined functions and consists of expressions alone. In this section, we explain the fundamental ideas of the flattening transformation based on a $\mathcal{CL}$ query.

For the purpose of our discussion, we strip down the $\mathcal{CL}$ running example Query Q1 and consider the following simplified variant that is sufficient to discuss all aspects.

```
[ ⟨o, sort [ ⟨l, o.od - l.sd⟩ | l ← ls, l.ok = o.ok ]⟩ | o ← os ]   (Q2)
```

To focus on nested iteration, we strip down this query even further and start with the following minimal variant:

```
[ [ l | l ← ls, l.ok = o.ok ] | o ← os ]                          (Q3)
```

General list comprehensions as defined in Section 1.4 for $\mathcal{CL}$ express (nested) iteration over multiple lists and filter lists according to guard expressions. While comprehensions are a convenient way to express query logic, they are not essential. We can replace the inner comprehension with a simple *iterator* in combination with a `restrict` combinator that filters the resulting list:

```
[ restrict [ ⟨l, l.ok = o.ok⟩ | l ← ls ] | o ← os ]               (Q4)
```

An iterator `[ `e_1 ` | x ← `e_2 ` ]` solely describes the evaluation of expression e_1 for all elements of collection e_2. Combinator `restrict` is a first-order variant of the usual higher-order `filter` combinator. It applies to lists of type `[⟨t,Bool⟩]` in which each element is paired with a Boolean value and retains only those elements for which it is `True`.

```
restrict : [⟨τ,Bool⟩] → [τ]
```

Compared to full comprehensions, the simpler form of iterators enables us to focus on the core aspect of nested iteration. We desugar comprehensions systematically in Section 4.1.

3.2.1 *Data Replication*

To describe nested iteration, we adopt the notion of *iteration scopes* from Grust [Gru05]. An expression e_1 in the head of an iterator `[ `e_1 ` | x ← `e_2 ` ]` is evaluated in an iteration scope s determined by the generator expression e_2. In Query Q4, the head of the outer iterator is evaluated in scope s_0 determined by the generator expression `os`. The head of the inner iterator is evaluated in iteration scope s_1 determined by the sequence of generator expressions `os` and `ls`:

$$\overbrace{\texttt{[restrict [} \underbrace{\texttt{⟨l, l.ok = o.ok⟩}}_{s_1} \texttt{ | l ← ls] | o ← os]}^{s_0}$$

An iteration scope can be described by the list in the corresponding generator expression. The outer head expression in iteration scope s_0 is evaluated for all elements of list `os`. The outer iteration thus is accurately described by `os`.

In contrast, list `ls` is not sufficient to describe the inner iteration in scope s_1: it is determined by both enclosing generators `o` $\leftarrow$ `os` and `l` $\leftarrow$ `ls`. For each `o`, the list `ls` is considered independently, conceptually as a separate copy of `ls`. The inner head expression in scope s_1 then is evaluated for each element of each copy of `ls`. To describe the shape of the inner iteration, we take this description verbatim and *replicate* `ls` for each element of `os`. We express replication with the combinator $\otimes$, typed as follows:

$$\otimes : \tau_1 \to [\tau_2] \to [\tau_1]$$

$e_1 \otimes e_2$ creates a list containing a copy of e_1 for each element of e_2:

$$e_1 \otimes e_2 \equiv [\ e_1 \mid \mathtt{x} \leftarrow e_2\ ]$$

Hence, expression `ls` $\otimes$ `os` has type $[[\tau_l]]$ and results in a nested list that contains copies of `ls`:

$$\mathtt{ls} \otimes \mathtt{os} = [\ \underbrace{[l_1, \ldots, l_m]}_{o_1}, \ldots, \underbrace{[l_1, \ldots, l_m]}_{o_n}\ ]$$

This nested list accurately describes the inner iteration scope s_1. The head expression of the innermost iterator is evaluated for each element of each inner list in `ls` $\otimes$ `os`.

3.2.2 *Lifted Combinators*

Let us for a moment assume that in scope s_1 only variable `l` occurs but not `o`. For the sake of brevity, we consider the following abstract form of Query Q4:

```
[ f [ g l | l ← ls ] | o ← os ]
```

The resulting list has the shape of `ls` $\otimes$ `os` and is computed as follows:

$$[\mathtt{f}\ [\mathtt{g}\ l_1, \ldots, \mathtt{g}\ l_m], \ldots, \mathtt{f}\ [\mathtt{g}\ l_1, \ldots, \mathtt{g}\ l_m]]$$

In the original query, combinators apply to single arguments at a time and iteration is expressed explicitly through iterators. If we can derive a *lifted* form $\mathtt{g}'$ of `g` that can be applied to `ls` $\otimes$ `os` and works on all elements in parallel, the inner iterator is no longer necessary. Likewise, a lifted variant $\mathtt{f}'$ of `f` that can be applied to $\mathtt{g}'$ `(ls` $\otimes$ `os)` allows to eliminate the outer iterator and transform the query into the following form:

$$\mathtt{f}'\ (\mathtt{g}'\ (\mathtt{ls} \otimes \mathtt{os}))$$

Lifted combinators are bulk operations that are evaluated in a *data-parallel* fashion on all elements of the input list. Consider the record selector `l.ok` in iteration scope s_1. A lifted record selector $e.\ell^1$ applies to a list of records:

$$e.\ell^1 \equiv [\ \mathtt{x}.\ell \mid \mathtt{x} \leftarrow e\ ]$$

However, as `l.ok` occurs in a *nested* iteration scope, we have *lists of lists* of arguments, namely `ls` $\otimes$ `os`. Accordingly, we assume lifted variant $e.\ell^2$ of record selection with the following meaning:

$$e.\ell^2 \equiv [\ [\ \mathtt{x}.\ell \mid \mathtt{x} \leftarrow \mathtt{xs}\ ] \mid \mathtt{xs} \leftarrow e\ ]$$

$e.\ell^2$ applies record selection to all inner elements of the argument list and does not change the shape of the list itself. With lifted operations $\langle \_, \_ \rangle^2$ (record construction) and the equality operator $=^2$ defined analogously, we can derive g' for the inner head expression in s_1, apply it to $\texttt{ls} \otimes \texttt{os}$ and thus eliminate the explicit inner iterator. We explain how to deal with the free occurrence of variable `o` in the next paragraph.

Expression $g'\ (\texttt{ls} \otimes \texttt{os})$ results in a list of type $[[\langle t_l, \texttt{Bool} \rangle]]$. The data-parallel variant f' of the outer head expression is derived easily by relying on a lifted $\texttt{restrict}^1$ combinator that applies to lists of arguments:

$$\texttt{restrict}^1\ e \equiv [\ \texttt{restrict xs} \mid \texttt{xs} \leftarrow e\ ]$$

Note that $\texttt{restrict}^1$ preserves the shape of the outer list and only modifies the inner lists, *i.e.* the actual arguments.

3.2.3 *Environment Lifting*

So far, we have conveniently ignored the free occurrence of variable `o` in scope s_1. The expression in scope s_1 can not be evaluated based on $\texttt{ls} \otimes \texttt{os}$ alone, though, as it provides only bindings for variable `l`.

The expression in scope s_0 is evaluated in an environment with a binding for variable `o`. A lifted variant of this expression evaluates all iterations at once. Conceptually, it is evaluated in a *parallel environment* that provides all bindings for variable `o` at once:

$$\texttt{o} \mapsto \texttt{os}$$

Likewise, the lifted variant of the expression in scope s_1 is evaluated in a parallel environment that provides all bindings for variable `l` at once:

$$\texttt{l} \mapsto \texttt{ls} \otimes \texttt{os}$$

However, we can't simply extend the parallel environment of s_1 with this binding for `o`. Values `os` and $\texttt{ls} \otimes \texttt{os}$ have different shape:

$$\begin{aligned} \texttt{l} &\mapsto [[l_1, \ldots, l_m], \ldots, [l_1, \ldots, l_m]] \\ \texttt{o} &\mapsto [\quad o_1 \quad, \ldots, \quad o_n \quad] \end{aligned}$$

To enable a uniform replacement of combinators p in scope s_1 with p^2, we bring the bindings to the same shape by *lifting* the binding for `o`: for each order, we create a copy of that order for each element of the corresponding list of lineitems:

$$\begin{aligned} \texttt{l} &\mapsto [\ [l_1, \ldots, l_m]\ , \ldots,\ [l_1, \ldots, l_m]\ ] \\ \texttt{o} &\mapsto [[\underbrace{o_1, \ldots, o_1}_{m}], \ldots, [\underbrace{o_n, \ldots, o_n}_{m}]] \end{aligned}$$

The lifted binding for `o` reflects that variable `o` is constant in s_1 for one particular evaluation of the inner iterator.

Is a new operator necessary to express per-element replication as above? Fortunately not: We can use the lifted version of operator $\otimes$:

$$\otimes^1 : [\tau_1] \to [[\tau_2]] \to [[\tau_1]]$$

With the lifted environment binding for `o`, we obtain the following parallel environment for s_1:

$$\begin{aligned} \texttt{l} &\mapsto \texttt{ls} \otimes \texttt{os} \\ \texttt{o} &\mapsto \texttt{os} \otimes^1 (\texttt{ls} \otimes \texttt{os}) \end{aligned}$$

```
   s0        s1
[ ⟨o, sort [ ⟨l,o.od - s.sd⟩ s2
           | l ← restrict [ ⟨l,l.ok = o.ok⟩
                          | l ← ls ] ]⟩
| o ← os ]
```

(a) Nested Iteration

```
let o = os
in ⟨o, sort¹ (let l = restrict¹ (let l = ls ⊗ o
                                     o = o ⊗¹ l
                                 in ⟨l,l.ok² =² o.ok²⟩²)
                  o = o ⊗¹ l
              in ⟨l,o.od² -² l.sd²⟩²)⟩¹
```

(b) Data-Parallel Operators

Figure 21: Query Q2 (Figure 21a) and its iterator-free lifted variant (Figure 21b).

Here, bindings for `o` and `l` have the same shape.

With the lifted parallel environment, the free occurrence of `o` in s_1 is no longer a problem and we can derive the lifted variant of the complete expression. The record selector `o.ok` can be replaced with $e.\ell^2$ applied to `os` $\otimes^1$ `(ls` $\otimes$ `os)`. The lifted comparison operator $=^2$ takes two lists of lists of arguments (here: `[[Int]]`), with both lists having the same shape:

$$\texttt{l.ok}^2 =^2 \texttt{o.ok}^2 \equiv \begin{matrix} [[l_1.\texttt{ok},\ldots,l_m.\texttt{ok}],\ldots,[l_1.\texttt{ok},\ldots,l_m.\texttt{ok}]] \\ \vdots \qquad\qquad \vdots \quad =^2 \quad \vdots \qquad\qquad \vdots \\ [[o_1.\texttt{ok},\ldots,o_1.\texttt{ok}],\ldots,[o_n.\texttt{ok},\ldots,o_n.\texttt{ok}]] \end{matrix}$$

Overall, we obtain the following lifted expression that solely relies on lifted combinators and is free of explicit iteration:

```
restrict¹ ( let l = ls ⊗ os
                o = os ⊗¹ (ls ⊗ os)
            in ⟨l, l.ok² =² o.ok²⟩² )
```

3.2.4 *Complete Running Example*

Proceeding in the same way as outlined above, we can derive a lifted variant free of iterators from the slightly more complex Query Q2. Figure 21a shows Query Q2 with the comprehension guard replaced by `restrict`. The three iterators define iteration scopes s_0, s_1 and s_2.

The lifted variant of Query Q2 in Figure 21b follows the same pattern as the previous example. For each iteration scope we construct a parallel environment that mimics the original lexical environment. Constant table references in the generator (here: `ls`) are replicated and names in the lexical environment of the iterator (here: `o`) are lifted to the appropriate shape. In this new environment, the head expression is lifted by replacing all combinators with their lifted variants.

The generators of both inner iterators are evaluated in iteration scope s_0. Note, that only the constant table reference `ls` is replicated, but not the `restrict` expression. We need a copy of `ls` for each iteration of scope s_0 in

order to evaluate all iterations of scope s_1 at once. In contrast, the generator expression for s_2 is lifted and its results already has the correct shape. For s_2, we still need to lift the binding for variable `o`, though. Although bindings for `l` and `o` in s_1 and s_2 have the same type, they are not identical: In s_2, `l` is a filtered version of `l` in s_1.

3.2.5 *Flat Data Parallelism*

For each iteration scope, we call the number of enclosing iterators its *iteration depth*. In Figure 21, scope s_0 is evaluated at depth 1 while s_1 and s_2 have depth 2. For scope s_0, we obtain a parallel environment with a binding for the iterator variable `o` of type `[Order]`. With flat lifted combinators p^1 we can evaluate all iterations of s_0 at once. At depth 2, replication adds one layer of list nesting for all environment bindings (`l : [[Lineitem]]` and `o : [[Order]]`). Lifted combinators p^2 operate on lists of lists of arguments. In general, at iteration depth d we have $d-1$ additional layers of list nesting on the original list type of generator expressions and use lifted combinators p^d. Consequently, for each combinator p, we require an infinite family of data-parallel variants p^d. For an operation p with type signature

$$p : \tau_1 \to \cdots \to \tau_n \to \tau$$

the type signature of p^d is

$$p^d : \overbrace{[\ldots[}^{d}\tau_1\overbrace{]\ldots]}^{d} \to \cdots \to \overbrace{[\ldots[}^{d}\tau_n\overbrace{]\ldots]}^{d} \to \overbrace{[\ldots[}^{d}\tau\overbrace{]\ldots]}^{d}$$

Note that $\mathcal{CL}$ allows not only scalar operations but also list combinators to be applied iteratively. Consequently, we also need lifted variants of list operations like `restrict`^{d} and `group`^{d}.

Are those families of lifted operators we have introduced actually essential? Fortunately not, due to a central insight of Blelloch and Sabot [BS89]: for any combinator p, all variants p^d with $d > 1$ can be mapped to p^1 statically. We require only flat lifted combinators p^1 that apply to flat lists of arguments. In Section 3.2.6, we will see that with a suitable representation of nested lists, this mapping does not incur any runtime cost.

We outline the mapping to flat lifted combinators with a concrete example from Query Q2. In the lifted variant of Figure 21b (Figure 21), expression `o.od`^{2} `-`^{2} `l.sd`^{2} computes the difference between the shipping date of all line items `l` and the order date of the corresponding orders `o`. Operator `-`^{2} is applied to two arguments of type `[[Date]]` and behaves as follows:

```
[[1995-11-09,1995-11-13,1995-12-16,1995-12-30] , [1993-02-19,1993-05-01]]
     -          -          -          -          -          -
[[1995-09-12,1995-09-12,1995-09-12,1995-09-12] , [1993-01-04,1993-01-04]]
     =          =          =          =          =          =
[[   65    ,   58     ,   62     ,   109    ] , [   46    ,   117    ]]
```

Both operands as well as the result have the same list structure. Combinator `-`^{2} computes the difference between pairs of corresponding `Date` values. The enclosing nested list structure is preserved in the result of the lifted combinator but is not relevant for the actual computation applied to the arguments. The family `-`^{d} of data-parallel operators differs only in the depth of the list structure that encloses the arguments. If we turn both operands into flat lists of type `[Date]` and apply `-`^{1}, we have essentially described the same computation — modulo list nesting.

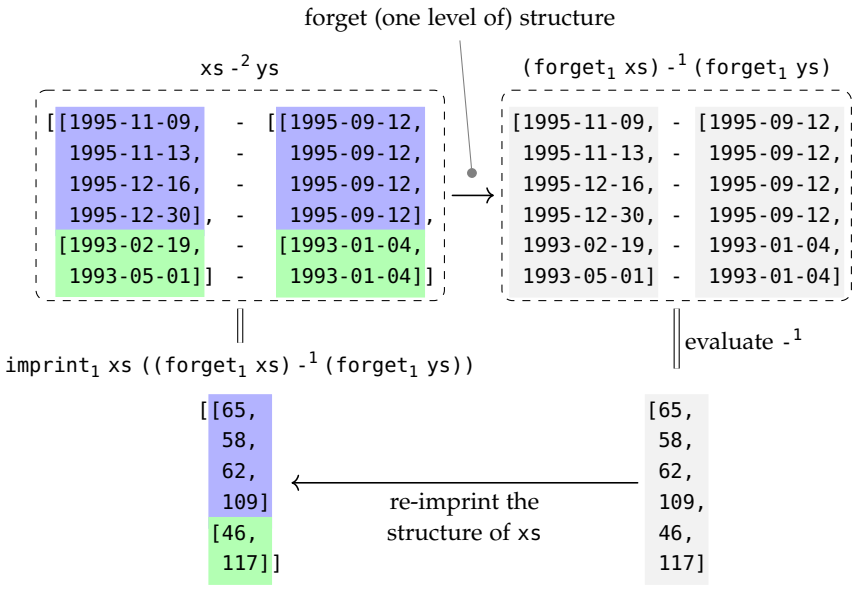


Figure 22: Restructuring nested lists with a flat representation.

This connection allows us to replace $\texttt{-}^2$ with $\texttt{-}^1$. We temporarily flatten the operands into lists of type `[Date]`, apply $\texttt{-}^1$ and subsequently restructure the result of type `[Int]` into a list of type `[[Int]]` whose structure is the same as that of the original operands.

To flatten and restructure lists, we introduce shape operations $\texttt{forget}_d$ and $\texttt{imprint}_d$ that modify only the shape of a list. Shape operations are typed as follows:

$$\frac{\Gamma \vdash e : \overbrace{[\ldots[}^{d}\,[t]]\ldots]}{\Gamma \vdash \texttt{forget}_d\ e : [t]} \qquad \frac{\Gamma \vdash e_1 : \overbrace{[\ldots[}^{d}\,[t_1]]\ldots] \qquad \Gamma \vdash e_2 : [t_2]}{\Gamma \vdash \texttt{imprint}_d\ e_1\ e_2 : \underbrace{[\ldots[}_{d}\,[t_2]]\ldots]}$$

Applied to an expression e, $\texttt{forget}_d\ e$ strips away the outer d layers of list nesting of e. In our example, applying $\texttt{forget}_1$ to both operands of $\texttt{-}^2$ results in operands of type `[Date]` that are suitable for $\texttt{-}^1$. Conversely, $\texttt{imprint}_1\ e_1\ e_2$ restructures the list e_2 according to the d outer list nesting layers of e_1. In Figure 22, we use $\texttt{forget}_1$ and $\texttt{imprint}_1$ to implement $\texttt{-}^2$.

With $\texttt{imprint}_d$ and $\texttt{forget}_d$ we can turn any lifted expression into an equivalent expression that uses only flat data parallelism in the form of p^1. Whenever a combinator p^d with $d > 1$ occurs, we use $\texttt{forget}_{d-1}$ to strip away the outer $d-1$ layers of list nesting from all operands and apply p^1 instead. We then restructure the resulting flat list of results with $\texttt{imprint}_{d-1}$ into the original nesting structure of the arguments.

3.2.6 *Flattening Data*

At this point, we can eliminate explicit iteration and nested data parallelism from $\mathcal{CL}$ queries. The underlying data model, however, is centered around nested lists. Although we strip layers of nesting from those lists temporarily with $\texttt{forget}_d$, the result of lifted combinators after $\texttt{imprint}_d$ is nested. Additionally, we strip only the list nesting as generated by nested iteration — the result of $\texttt{forget}_d$ is not in general a flat list of scalar values. Lifted

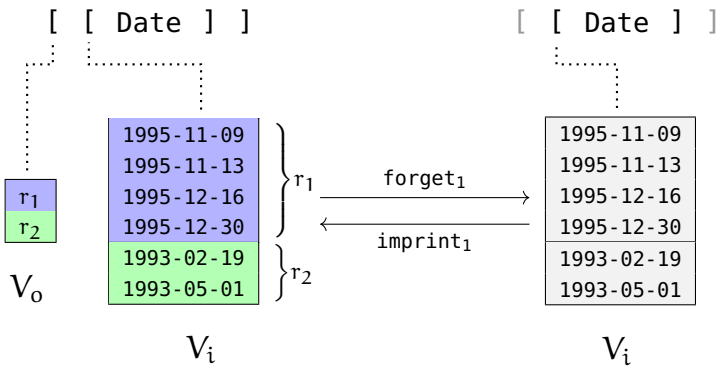


Figure 23: Shape operations on segment vectors.

list combinators like `sort`^{1} are defined on nested lists. However, we target query engines that support flat collections only and need a flat representation of nested lists.

After lifting, $\mathcal{CL}$ queries are littered with shape operations: each application of an operation in a nested iteration incurs applications of $\texttt{forget}_\text{d}$ and $\texttt{imprint}_\text{d}$. The flat data representation has to support `forget` and `imprint` that work on the list *structure* as well as lifted operations that work on list *content* efficiently.

With a representation of nested lists that separates structure and content, we can achieve both objectives. Figure 23 depicts the representation of the nested list of type `[[Date]]` from Figure 22. The nested list is encoded in two flat *vectors* V_o and V_i. The inner vector V_i stores the content for both inner lists. Its elements are partitioned into *segments* r_1 and r_2, one for each inner list. The outer vector V_o encodes the structure of the outer list: it has two elements, each of which represents one of the inner lists and references the corresponding segment in V_i.

As sketched in Figure 23, evaluating $\texttt{forget}_\text{d}$ and $\texttt{imprint}_\text{d}$ on segment vectors involves no work at all. The representation of the flat list of type `[Date]` is obtained from V_o and V_i with $\texttt{forget}_1$ merely by ignoring the outer vector V_o and the segment structure of the inner vector. The operands to `-`^{1} are then two unstructured vectors of `Date` values on which the data-parallel subtraction can be evaluated easily for each pair of corresponding values. The resulting vector has the same shape as the original vectors. Restructuring with $\texttt{imprint}_1$ is achieved by combining the result vector with the original outer vector V_o. Hence, both $\texttt{forget}_\text{d}$ and $\texttt{imprint}_\text{d}$ are compile-time operations with no runtime cost.

This description of segment vectors is deliberately vague. In particular, we have not yet defined how vector segments are encoded. In Section 4.3 we discuss concrete encodings of vectors and segments and devise a flat representation of nested lists that fits the processing model of query engines.

3.3 RELATED WORK AND OUTLOOK

The transformation of $\mathcal{CL}$ queries that we have sketched in this chapter is not original at all. We simply apply principles of the flattening transformation as originally described by Blelloch and Sabot [BS89] to our query language $\mathcal{CL}$.

We have translated a nested query with nested iteration into an iterator-free query on flat collections (for now: lists) that relies on flat data-parallel

operators. Essentially, we have performed query flattening. To execute flattened code, an engine primarily has to provide flat data-parallel operators. Most uses of the flattening transformation target explicitly parallel hardware like vector processors and GPUs ([Ble+94], [BR12]) as well as concurrent [Pey+08] and distributed [Kel99] runtimes. These are not the only choices, however. Relational query engines are geared towards the efficient execution of algebraic bulk operators — essentially flat data parallelism. Lifted versions of scalar operations like $+^1$, for instance, are readily provided by projection. In the rest of this thesis, we explore that analogy and show that it extends to collection combinators (*e.g.* $\texttt{sort}^1$) as well.

We are not the first to draw the connection between query flattening and the flattening transformation. Some query flattening approaches discussed in Chapter 2 — *Loop-Lifting*, *Dodo* and the translation of Suciu [Suc97] — are structurally quite similar to the translation sketched here. *Loop-Lifting* and *Dodo* seem to have been developed independently, though. Employing the flattening transformation for query flattening has been suggested by Suciu [ST94; Suc95; Suc96], Cheney *et al.* [CLW14a], Mayr [May13] and Rittinger [Rit11].

Our goal is to show that the flattening transformation can lower an expressive, nested query language into a form that can be executed by regular relational query engines. We will describe a version of the flattening transformation that takes the specific aspects of query processing into account. This gives us an alternative account of query flattening (Chapter 2) that is more structured and comprehensible than previous work.

4 FLATTENING QUERIES

With the overview in Chapter 3, we have outlined query flattening based on the flattening transformation. In this chapter, we describe the approach in detail. We write *Query Flattening* for our specific approach to distinguish it from the general term of query flattening as discussed in Chapter 2.

Conforming with established principles [Sha+16], we describe *Query Flattening* as a sequence of lowering steps between intermediate languages. Each subsequent step strips away one layer of abstraction and translates to a simpler target language. Compared to a monolithic translation (*e.g. Loop-Lifting*), each individual lowering step is easier to comprehend. Additionally, as we do not strip all layers at once, we can delay details of the data representation (*e.g.* the representation of order) and consider them independently. The pipeline of lowerings is depicted in Figure 24.

Query Flattening incorporates specifics of query processing but does not prescribe a particular *backend*. Backend here denotes a query engine that offers bulk operations on flat collections. *Query Flattening* targets the flat language $\mathcal{SL}$ which can serve as the starting point for code generation for a variety of backends. In a subsequent chapter, we consider one concrete backend and describe the generation of efficient relational queries on flat, unordered multisets. This paves the way for the generation of SQL:2003 queries that can be executed on off-the-shelf relational database systems. We discuss alternative implementations of $\mathcal{SL}$ in Section 4.3.2.3.

The fine-grained nature of *Query Flattening* allows us to plug in optimizations after any lowering step. Meaningful optimizations can be performed before lowering to a specific backend. All potential backends profit from such optimizations. Concretely, the pipeline of Figure 24 includes one optimization step on $\mathcal{SL}$ that focuses on scalar expressions. Further optimizations are discussed in Chapter 5.

The actual lowering steps are the following:

(1) As a preparatory step, *desugaring* (Section 4.1) simplifies $\mathcal{CL}$ queries. General comprehensions are replaced with iterators. The target language $\mathcal{CL}_{\mathrm{d}}$ is centered around iterators and ensures that all combinators are applied iteratively.

(2) *Lifting* (Section 4.2) eliminates iteration and thus nested data parallelism. Iterators are replaced with lifted combinators and replication. The target language $\mathcal{FL}$ is an algebraic language without iterators based on nested lists.

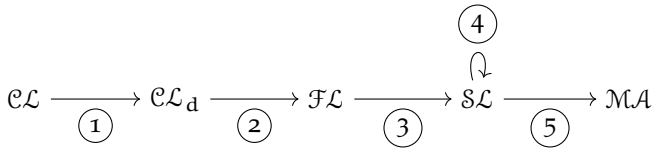


Figure 24: Pipeline of lowering steps that define *Query Flattening*.

(3) *Shredding* (Section 4.3) eliminates nested lists and lowers operations on nested lists to operations on flat vectors. It targets $\mathcal{SL}$, a language of simple vector operators.

(4) $\mathcal{SL}$ vector operators are fused to merge scalar operators and avoid excessive intermediate results.

(5) A backend that eliminates implicit order of lists and vectors and emits relational queries on unordered multisets is described in Chapter 6. $\mathcal{SL}$ operators are lowered to the relational algebra $\mathcal{MA}$ on multisets.

As *Loop-Lifting* (Section 2.2.2), *Query Flattening* does not depend on preceding normalization and can handle any type-correct $\mathcal{CL}$ expression. However, while normalization is not strictly necessary, normalization is nevertheless beneficial. We explore this topic in Chapter 5.

We restrict admissible $\mathcal{CL}$ queries in one regard: We consider only $\mathcal{CL}$ queries $e : [\tau]$ that return lists. Top-level scalar values are not allowed. This restriction is only natural in the context of query flattening. After all, we target query engines whose data model typically is restricted to collections and does not allow top-level scalar values. The expressiveness of the query language is not limited notably. For queries that do return scalar values (*e.g.* TPC-H Q4), the result can be wrapped in a singleton list.

4.1 DESUGARING COMPREHENSIONS

We rewrite list-typed $\mathcal{CL}$ expressions $e : [\tau]$ into the dialect $\mathcal{CL}_d$ defined in Figure 25. $\mathcal{CL}_d$ differs in two aspects from $\mathcal{CL}$:

1. Instead of proper comprehensions with multiple qualifiers, $\mathcal{CL}_d$ supports only simple iterators $[\ e_1 \mid x \leftarrow e_2\ ]$ with exactly one generator and no guards. Guards are replaced with the `restrict` combinator described in Section 3.2.

2. The form of top-level expressions is restricted to iterators with an optional application of `concat`. The generator expression may only be a literal list or a table reference.

In $\mathcal{CL}_d$, there are only two forms of expressions which can occur outside of the head of an iterator and are not evaluated iteratively: List-typed constants (literal lists $[v, \ldots, v]$ and table references `table(t)`), applications of `concat` and iterators $[\ e \mid x \leftarrow e\ ]$ themselves.

Considering only $\mathcal{CL}_d$ simplifies subsequent lowering steps considerably. We profit particularly from the second restriction: With the exception of `concat`, combinators are only applied iteratively. With iterators eliminated, we only have to provide lifted combinators and can omit the not-lifted version, leading to a simple and uniform translation.

In the following, we describe rewrite rules that transform any $\mathcal{CL}$ query that returns a list into an equivalent $\mathcal{CL}_d$ query. We write $e_1 \rightsquigarrow e_2$ for a rule that rewrites expression e_1 into expression e_2.

SUBSTITUTION ON COMPREHENSIONS One technicality has to be observed when applying rewrite rules to comprehensions. Here and in following chapters (in particular Chapter 5) we transform $\mathcal{CL}$ queries according to rewrite rules. Rewrite rules substitute expressions for variables in a given

Top-Level Queries

$q ::= \texttt{concat [} e \texttt{ | } x \leftarrow ds \texttt{] } \mid \texttt{[} e \texttt{ | } x \leftarrow ds \texttt{]}$

Expressions, Iterators

$ds ::= \texttt{table(t)} \mid \texttt{[}v\texttt{, ..., }v\texttt{]}$

$e ::= x \mid v \mid ds \mid \texttt{let } x \texttt{ = } e \texttt{ in } e \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \mid \texttt{[} e \texttt{ | } x \leftarrow e \texttt{]}$
$\quad \mid p\ e \cdots e$

Built-In Operations

$p ::= \texttt{...} \mid \texttt{restrict}$

Figure 25: Grammar of language $\mathcal{CL}_d$ with iterators instead of comprehensions. Combinators p are as in Figure 8 extended with `restrict`.

expression. We use the usual notation $e[e'/x]$ for the capture-avoiding substitution of e' for x in e. To prevent free variables in the substitute e' from being captured, capturing bindings are alpha-renamed.

On comprehensions, we often apply rewrites in the middle of a qualifier list. We extend substitution to qualifier lists and write $qs[e'/x]$ for the capture-avoiding substitution of e' for x in all qualifiers of qs. If necessary, generators in qs are renamed to avoid capturing variables that occur free in e'. Note though that alpha-renaming performed in the qualifier list has to be performed in the head expression as well. Consider the following expression:

$$\texttt{[} e \texttt{ | } qs[\texttt{y.1}/x] \texttt{]} \quad \text{with } qs \equiv y \leftarrow e_1\texttt{ , } z \leftarrow e_2$$

The generator binding `y` will be renamed to avoid capturing `y` in the substitute. This will change the binding for `y` in the head expression, though. To be correct, substitution in the qualifier list has to rewrite e as well. In terms of notation, this is inconvenient, however. In examples and rewrite rules in this thesis, we ignore this issue and assume that no renaming is necessary.

ELIMINATE COMPREHENSIONS Comprehensions are desugared by two rules that match a pattern at the beginning of a qualifier list. We assume that the qualifier list does not start with a guard — if it does, the guard expression is pushed back in the qualifier list first. Given a generator $x \leftarrow e_2$ and a guard e_3, Rule DESUGAR-PRED merges both qualifiers into an application of the `restrict` combinator.

$$\begin{array}{ll} \texttt{[} e_1 \texttt{ | } x \leftarrow e_2\texttt{ , } e_3\texttt{ , } qs \texttt{]} & \\ \quad \rightsquigarrow & \text{(DESUGAR-PRED)} \\ \texttt{[} e_1 \texttt{ | } x \leftarrow \texttt{restrict [} \langle x, e_3 \rangle \texttt{ | } x \leftarrow e_2 \texttt{]}\texttt{ , } qs \texttt{]} & \end{array}$$

Given two generators $x \leftarrow e_2$ and $y \leftarrow e_3$, Rule DESUGAR-GENS computes pairs of all elements of e_2 and e_3 in the correct order with nested iterators. Record selectors replace the generator variables x and y. Here, our remarks about substitution in a qualifier list are relevant: we choose the new gener-

ator variable z as a globally fresh variable that will not be captured by any generator in qs.

$$
\begin{aligned}
&[\ e_1 \mid x \leftarrow e_2, y \leftarrow e_3, qs\ ] \\
&\quad \rightsquigarrow \\
&[\ e_1[z.1/x][z.2/y] \qquad \text{(DESUGAR-GENS)} \\
&\mid z \leftarrow \texttt{concat}\ [\ [\ \langle x, y\rangle \mid y \leftarrow e_3\ ] \mid x \leftarrow e_2\ ] \\
&, qs[z.1/x][z.2/y]\ ]
\end{aligned}
$$

Applied exhaustively, Rules DESUGAR-PRED and DESUGAR-GENS turn all comprehensions into iterators of the form $[\ e_1 \mid x \leftarrow e_2\ ]$. Starting at the beginning of the qualifier list, all qualifiers are merged into one.

Comprehension guards are usually desugared into conditionals [Wad90]. Query Q3 would be desugared as follows:

```
[ concat [ if l.ok = o.ok then sng l else [] | l ← ls ] | o ← os ]   (Q5)
```

However, we solely care about comprehensions over collection data structures. The query engines we target offer efficient primitives for filtering collections. Translating guards as conditionals would complicate intermediate representations of a query and blur the query structure. Recovering the actual collection filter from the translated conditional might fail and lead to inefficient backend code. Instead, we translate directly to the combinator `restrict` that can be easily mapped to a backend primitive. This resembles the desugaring scheme of Fegaras and Maier [FM00].

TOP-LEVEL EXPRESSIONS After desugaring, we rewrite any top-level expression that does not match the syntactic category q of Figure 25. Any closed, list-typed expression $e : [\tau]$ can be rewritten into $\mathcal{CL}_d$ by wrapping it in an iterator with a singleton generator:

$$e \rightsquigarrow \texttt{concat}\ [\ e \mid z \leftarrow [\langle\rangle]\ ] \qquad \text{(DESUGAR-TOP)}$$

We apply the rewrite only to expressions that are not already valid top-level queries.

Note that Rule DESUGAR-TOP makes the dummy iteration encoded in *Loop-Lifting*'s *loop* relation explicit.

4.1.1 *Definition of the Meta-Language*

Throughout this thesis, we use a number of intermediate languages to describe the lowering of $\mathcal{CL}$ to relational algebra. For each of them, we define a denotational semantics in terms of lists and records. We define semantics using a Haskell-like meta-language $\mathcal{ML}$. In fact, we would prefer to define *executable* semantics in terms of actual Haskell code. However, Haskell's lack of records is inconvenient.

Meta-level lists with elements x_1 to x_n are written as $[x_1, \ldots, x_n]$, with $[]$ being the empty list. The concatenation of lists *xs* and *ys* is written as $xs +\!\!+ ys$. Lists in *insert* notation are defined with $x : xs = [x] +\!\!+ xs$. In $\mathcal{ML}$, we rely heavily on list comprehensions with the usual syntax and semantics [PW07].

We use a number of list combinators in $\mathcal{ML}$:

- List elements can be enumerated with enum:

$$\text{enum}\ [x_1, \ldots, x_n] = [\langle x_1, 1\rangle, \ldots, \langle x_n, n\rangle]$$

- Lists are sorted with sortWith:

$$\mathsf{sortWith}\ (\lambda x.x.2)\ [\langle 5,7\rangle, \langle 6,3\rangle, \langle 4,3\rangle] = [\langle 6,3\rangle, \langle 4,3\rangle, \langle 5,7\rangle]$$

  Note that sortWith performs stable sorting: elements in the input list that have a tie in their sorting key appear in the original order.

- Folding and scanning of lists is defined using pattern matching:

$$\begin{array}{lcl} \mathsf{scan\ f\ z\ []} & = & \mathsf{[]} \\ \mathsf{scan\ f\ z\ (x : xs)} & = & \mathsf{f\ z\ x : scan\ f\ (f\ z\ x)\ xs} \end{array}$$

$$\begin{array}{lcl} \mathsf{foldl\ f\ z\ []} & = & \mathsf{z} \\ \mathsf{foldl\ f\ z\ (x : xs)} & = & \mathsf{foldl\ f\ (f\ z\ x)\ xs} \end{array}$$

  For example, $\mathsf{foldl}\ (+)\ 0\ [1,2,3] = 6$ and $\mathsf{scan}\ (+)\ 0\ [1,2,3] = [1,3,6]$.

- Combinator groupWith groups a list based on a function that projects a *grouping key* for each element:

$$\mathsf{groupWith}\ (\lambda x.x.2)\ [\langle 5,7\rangle, \langle 6,3\rangle, \langle 4,3\rangle] = [\langle 3, [\langle 6,3\rangle, \langle 4,3\rangle]\rangle, \langle 7, [\langle 5,7\rangle]\rangle]$$

  Note that groupWith includes the grouping key for each group. In each group, the relative order of the input list is preserved.

- Duplicates in a list are eliminated with nubWith:

$$\mathsf{nubWith}\ (\lambda x.x.2)\ [\langle 5,7\rangle, \langle 6,3\rangle, \langle 4,3\rangle] = [\langle 5,7\rangle, \langle 6,3\rangle]$$

  Note that nubWith preserves the order of the input list. Out of multiple elements with the same key, nubWith keeps the first.

4.1.2 *Indexed Semantics of $\mathcal{CL}_d$*

We define the semantics of $\mathcal{CL}_d$ in terms of lists and records. We interpret lists as *indexed* lists in the same way as the annotated semantics specified by Cheney *et al.* [CLW14a] for *Query Shredding*. A list of type $[\tau]$ is interpreted as an indexed list of type $[\langle k{:}\delta, p{:}\tau\rangle]$. In an indexed list $[\langle k = k_1, p = x_1\rangle, \ldots, \langle k = k_n, p = x_n\rangle]$ of type $[\langle k{:}\delta, p{:}\tau\rangle]$, each element x_i is annotated with a unique *scalar* index k_i of type δ that denotes its identity.

The indexed semantics of $\mathcal{CL}_d$ serves two purposes. First, it is easier to specify some operations with indexes — in particular the shape operations `forget` and `imprint` in Section 4.2.4. Primarily, though, it provides a connection to the flat representation of nested lists defined in Section 4.3. In the remainder of this thesis, we use an index-based flat representation of nested lists as in prior work on query flattening (Chapter 2). If we were to define the semantics of intermediate languages based on list positions to identify elements, it would be hard to relate both worlds. Once we lower our flat representations to an *unordered* flat representation, positional access is not an option anymore. Additionally, by decoupling *Query Flattening* from list order, it provides the basis for extending *Query Flattening* to queries over unordered collections.

Base values v are interpreted as $[\![v]\!]$ and base operators $c(\_)$ as $[\![c]\!](\_)$. The interpretation of scalar expression s is expressed as $[\![s]\!]$ which is a direct mapping to $\mathcal{ML}$. We write ρ for an environment that maps variables to

values. Looking up a variable x is written as $\rho(x)$ and extension of the environment with a variable x as $\rho\,[x \mapsto v]$.

Function $\mathcal{I}[\![q]\!]_{\{\}}$ interprets a top-level $\mathcal{CL}_d$ expression q. We let ρ range over an environment that maps variables to values. The interpretation of basic constructs is trivial, then.

$$\mathcal{I}[\![x]\!]_\rho = \rho(x) \qquad (\mathcal{CL}_d\text{-VAR})$$

$$\mathcal{I}[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!]_\rho = \mathcal{I}[\![e_2]\!]_{\rho[x \mapsto \mathcal{I}[\![e_1]\!]_\rho]} \qquad (\mathcal{CL}_d\text{-VAR})$$

$$\mathcal{I}[\![e.\ell]\!]_\rho = \mathcal{I}[\![e]\!]_\rho.\ell \qquad (\mathcal{CL}_d\text{-REC-SEL})$$

$$\mathcal{I}[\![\langle \ell_i = e_i \rangle_{i=1}^n]\!]_\rho = \langle \ell_i = \mathcal{I}[\![e_i]\!]_\rho \rangle_{i=1}^n \qquad (\mathcal{CL}_d\text{-VAR})$$

$$\mathcal{I}[\![v]\!]_\rho = [\![v]\!] \qquad (\mathcal{CL}_d\text{-LIT})$$

$$\mathcal{I}[\![c(e_i)_{i=1}^n]\!]_\rho = [\![c]\!](\mathcal{I}[\![e_i]\!]_\rho)_{i=1}^n \qquad (\mathcal{CL}_d\text{-BASEOP})$$

In $\mathcal{CL}_d$, lists are originally obtained from literal lists, table references or singleton lists. For these three constructs, we have to provide initial indexes. Indexes for literal lists are derived from an enumeration of the elements. A table t is interpreted as a list $[\![t]\!]$ with elements in some arbitrary order. We impose a canonical order on tables by sorting them according to the primary key. For each element x of $[\![t]\!]$, $\mathrm{pk}_t(x)$ returns the scalar record of the primary key. The primary key also defines the initial indexes. We uniformly use the primary key although any candidate key would be sufficient.

$$\mathcal{I}[\![[v_1, \ldots, v_n]]\!]_\rho = [\langle k = 1, p = [\![v_1]\!]\rangle, \ldots, \langle k = n, p = [\![v_n]\!]\rangle] \qquad (\mathcal{CL}_d\text{-LIST})$$

$$\mathcal{I}[\![\texttt{table(t)}]\!]_\rho = [\ \langle k = \mathrm{pk}_t(x), p = x\rangle \mid x \leftarrow \mathsf{sortWith}\ (\lambda x.\, \mathrm{pk}_t(x))\ [\![t]\!]\ ] \qquad (\mathcal{CL}_d\text{-TABLE})$$

$$\mathcal{I}[\![\mathsf{sng}\ e]\!]_\rho = [\langle k = \langle\rangle, p = \mathcal{I}[\![e]\!]_\rho\rangle] \qquad (\mathcal{CL}_d\text{-SNG})$$

Iterators and list combinators are directly interpreted by their counterparts in the meta-language. Maintaining the indexes of input lists requires some gymnastics with records.

$$\mathcal{I}[\![[\ e_1 \mid x \leftarrow e_2\ ]]\!]_\rho = [\ \langle k = x.k, p = \mathcal{I}[\![e_1]\!]_{\rho[x \mapsto x.p]}\rangle \mid x \leftarrow \mathcal{I}[\![e_2]\!]_\rho\ ] \qquad (\mathcal{CL}_d\text{-ITERATOR})$$

$$\mathcal{I}[\![\texttt{reduce\{}s_z\texttt{, }s_f\texttt{\}}\ e]\!]_\rho = \mathsf{foldl}\ [\![s_f]\!]\ [\![s_z]\!]\ \mathcal{I}[\![e]\!]_\rho \qquad (\mathcal{CL}_d\text{-REDUCE})$$

$$\mathcal{I}[\![\texttt{number } e]\!]_\rho = [\ \langle k = x.1.k, p = \langle x.1.p, x.2\rangle\rangle \mid x \leftarrow \mathsf{enum}\ \mathcal{I}[\![e]\!]_\rho\ ] \qquad (\mathcal{CL}_d\text{-NUMBER})$$

$$\mathcal{I}[\![\texttt{sort } e]\!]_\rho = [\ \langle k = x.k, p = x.p.2\rangle \mid x \leftarrow \mathsf{sortWith}\ (\pi_2 \circ \pi_p)\ \mathcal{I}[\![e]\!]_\rho\ ] \qquad (\mathcal{CL}_d\text{-SORT})$$

$$\mathcal{I}[\![\texttt{distinct } e]\!]_\rho = \mathsf{nubWith}\ (\pi_2 \circ \pi_p)\ \mathcal{I}[\![e]\!]_\rho \qquad (\mathcal{CL}_d\text{-DISTINCT})$$

$$\mathcal{I}[\![\texttt{concat } e]\!]_\rho = [\ \langle k = \langle xs.k, x.k\rangle, p = x.p\rangle \mid xs \leftarrow \mathcal{I}[\![e]\!]_\rho, x \leftarrow xs.p\ ] \qquad (\mathcal{CL}_d\text{-CONCAT})$$

$$\mathcal{I}[\![\texttt{restrict } e]\!]_\rho = [\ \langle k = x.k, p = x.p.1\rangle \mid x \leftarrow \mathcal{I}[\![e]\!]_\rho, x.p.2\ ] \qquad (\mathcal{FL}\text{-RESTRICT})$$

$$\mathcal{I}[\![\texttt{group } e]\!]_\rho = [\ \langle k = kg.1, p = \langle kg.1, [\ \langle k = g.k, p = g.p.1\rangle \mid g \leftarrow kg.2\ ]\rangle\rangle \mid kg \leftarrow \mathsf{groupWith}\ (\pi_2 \circ \pi_p)\ \mathcal{I}[\![e]\!]_\rho\ ] \qquad (\mathcal{CL}_d\text{-GROUP})$$

$n ::= \langle \text{positive natural number } 1, 2, \ldots \rangle$

Expressions

$e ::= x \mid [v, \ldots, v] \mid \texttt{table(t)} \mid \texttt{let } x = e \texttt{ in } e \mid p\ e \cdots e$
$\mid o\ e \cdots e \mid e \otimes e \mid \texttt{rep}\{v\}\ e \mid \texttt{forget}_n\ e \mid \texttt{imprint}_n\ e\ e \mid$

Combinators

$o ::= \texttt{restrict} \mid \texttt{concat} \mid \texttt{combine}$

Lifted Combinators

$p ::= \texttt{sort}^\uparrow \mid \#^\uparrow \mid \texttt{concat}^\uparrow \mid \texttt{distinct}^\uparrow \mid \texttt{group}^\uparrow \mid \texttt{append}^\uparrow$
$\mid \langle \ell = \_, \ldots, \ell = \_ \rangle^\uparrow \mid \texttt{c}(\_, \ldots, \_)^\uparrow \mid \_.\ell^\uparrow \mid \texttt{restrict}^\uparrow \mid \_ \otimes^\uparrow \_$
$\mid \texttt{combine}^\uparrow \mid \texttt{sng}^\uparrow \mid \texttt{reduce}\{s, s\}^\uparrow$

Figure 26: Syntax of language $\mathcal{FL}$ that trades iterators for lifted combinators.

The only list combinator with two inputs so far is `append` which requires special attention. Indexes are non-uniform and in general the indexes of both `append` operands do not have the same type. Hence, we provide new uniform indexes of type `Int` for both inputs by enumerating the elements. These new indexes are combined with `Int` tags 1 and 2 to make them unique in the result list.

$$\begin{aligned}\mathcal{I}[\![\texttt{append } e_1\ e_2]\!]\rho = {} & [\ \langle \texttt{k} = \langle 1, \texttt{x.2} \rangle, \texttt{p} = \texttt{x.1.p} \rangle \mid \texttt{x} \leftarrow \texttt{enum } \mathcal{I}[\![e_1]\!]\rho\ ] \\ & +\!\!+ \\ & [\ \langle \texttt{k} = \langle 2, \texttt{x.2} \rangle, \texttt{p} = \texttt{x.1.p} \rangle \mid \texttt{x} \leftarrow \texttt{enum } \mathcal{I}[\![e_2]\!]\rho\ ]\end{aligned} \qquad (\mathcal{FL}\text{-APPEND})$$

4.2 LIFTING: FLATTENING NESTED DATA-PARALLELISM

In Section 3.2 and Section 3.2.5, we have outlined the elimination of iterators based on data replication, environment lifting and lifted combinators. Here, we define the translation from $\mathcal{CL}_d$ with iterators into the algebraic language $\mathcal{FL}$ with bulk operators precisely

4.2.1 *Flat Data-Parallel Language*

Lifting targets the language $\mathcal{FL}$ whose syntax is defined in Figure 26. Compared to $\mathcal{CL}_d$, it trades iteration for combinators that implement replication ($\otimes$, $\texttt{rep}\{v\}$), lifted combinators ($p^\uparrow$) and environment lifting ($\otimes^\uparrow$) as well as shape operators $\texttt{imprint}_d$ and $\texttt{forget}_d$. Typing rules for $\mathcal{FL}$ are defined in Figure 27. Languages $\mathcal{CL}$ and $\mathcal{FL}$ share the same data model: arbitrary combinations of record and list type constructors.

In Chapter 3 we use lifted combinators p^d at iteration depth d. These combinators are subsequently normalized to p^1. Combinators p^d with $d > 1$, however, are not necessary in an explicit form. Language $\mathcal{FL}$ includes only flat data-parallel operators $p^\uparrow$ that are equivalent to p^1.

Combinator `combine` has the following type:

```
combine : [Bool] → [τ] → [τ] → [τ]
```

It merges two lists of the same type according to a list of Boolean flags:

```
combine [True,True,False,True,False] [1, 2, 3] [7, 8] = [1,2,7,3,8]
```

$$\frac{\Gamma \vdash e : \overbrace{[\ldots[}^{d}\,[\tau]]\ldots]}{\Gamma \vdash \texttt{forget}_d\ e : [\tau]}\ \mathcal{FL}\text{-TY-FORGET}$$

$$\frac{\Gamma \vdash e_1 : \overbrace{[\ldots[}^{d}\,[\tau_1]]\ldots] \qquad \Gamma \vdash e_2 : [\tau_2]}{\Gamma \vdash \texttt{imprint}_d\ e_1\ e_2 : \underbrace{[\ldots[}_{d}\,[\tau_2]]\ldots]}\ \mathcal{FL}\text{-TY-IMPRINT}$$

$$\frac{\Gamma \vdash e_1 : [\delta] \qquad \Gamma \vdash e_2 : [\tau]}{\Gamma \vdash e_1 \otimes e_2 : [[\delta]]}\ \mathcal{FL}\text{-TY-REPLICATE}$$

$$\frac{\vdash \nu : \pi \qquad \Gamma \vdash e : [\tau]}{\Gamma \vdash \texttt{rep}\{\nu\}\ e : [\pi]}\ \mathcal{FL}\text{-TY-REPLICATE-BASE}$$

$$\frac{\Gamma \vdash e_1 : [\tau_1] \qquad \Gamma \vdash e_2 : [[\tau_2]]}{\Gamma \vdash e_1 \otimes^{\uparrow} e_2 : [[\tau_1]]}\ \mathcal{FL}\text{-TY-REPLICATE-LIFT}$$

$$\frac{\Sigma(p) = \tau_1 \to \cdots \to \tau_n \to \tau \qquad [\Gamma \vdash e_i : [\tau_i]\,]_{i=1}^{n}}{\Gamma \vdash p^{\uparrow}\ e_1\ \cdots\ e_n : [\tau]}\ \mathcal{FL}\text{-TY-OP-LIFT}$$

$$\frac{\Sigma(o) = \tau_1 \to \cdots \to \tau_n \to \tau \qquad [\Gamma \vdash e_i : \tau_i]_{i=1}^{n}}{\Gamma \vdash o\ e_1\ \cdots\ e_n : \tau}\ \mathcal{FL}\text{-TY-OP}$$

Figure 27: Typing rules for $\mathcal{FL}$. Typing rules for variables, `let`-bindings, table references and literal lists are as in Figure 9.

Next to the replication operators $\otimes$ and $\texttt{rep}\{\nu\}$, only few non-lifted combinators are included. Due to the restrictions of $\mathcal{CL}_d$, `concat` is the only combinator that can appear outside of iteration. The non-lifted forms of `restrict` and `combine` are necessary to lift `if`-conditionals.

Replication of data is implemented by operators $\otimes$ and $\texttt{rep}\{\nu\}$. Both behave essentially the same: A value is replicated for each element of a list.

$$\begin{aligned} e_1 \otimes e_2 &= [\ e_1 \mid x \leftarrow e_2\ ] \\ \texttt{rep}\{\nu\}\ e &= [\ \nu \mid x \leftarrow e\ ] \end{aligned}$$

The difference between both is apparent in typing rules $\mathcal{FL}$-TY-REPLICATE and $\mathcal{FL}$-TY-REPLICATE-BASE. In contrast to the type of $\otimes$ defined in Section 3.2, $\otimes$ replicates not arbitrary values but is restricted to lists. Combinator $\texttt{rep}\{\nu\}$ replicates a fixed atomic literal ν.

In $\mathcal{FL}$, literal values are only included in the form of literal lists $[\nu, \ldots, \nu]$. In contrast to $\mathcal{CL}$, atomic literals ν are not allowed as expressions. They can only be expressed with operator $\texttt{rep}\{\nu\}$ that expands ν to a list. Operations with scalar results (*e.g.* `sum`, `+`) only appear in lifted form and produce lists. As a consequence, $\mathcal{FL}$ can not express a purely scalar computation. All $\mathcal{FL}$ expressions are list-typed. This property will come in handy in Section 4.3.

Note that $\mathcal{FL}$ is essentially a variable-free language, hence an *algebra*. Variables are only bound by `let`-expressions that are included for convenience but are not essential to the semantics of the language. We translate the complex-object calculus $\mathcal{CL}$ into the algebraic language $\mathcal{FL}$. This step is the equivalent of translating into point-free nested relational algebras in Chapter 2.

4.2.2 *Translating to $\mathcal{FL}$*

We specify the lowering from $\mathcal{CL}_d$ to $\mathcal{FL}$ as a syntax-directed function $\mathcal{L}[\![-]\!]$. A $\mathcal{CL}_d$ top-level expression $q : [\tau]$ lowers to a $\mathcal{FL}$ expression $\mathcal{L}[\![q]\!] : [\tau]$.

TOP-LEVEL EXPRESSIONS Let us begin with top-level expressions that are not enclosed by an iterator. In $\mathcal{CL}_d$, top-level expressions can only be applications of `concat`, list constants and iterators.

For top-level list constants and `concat` applications, $\mathcal{L}[\![-]\!]$ is merely the identity:

$$\mathcal{L}[\![\texttt{concat}\ e]\!] = \texttt{concat}\ \mathcal{L}[\![e]\!] \qquad \text{(LIFT-CONCAT-TOP)}$$

$$\mathcal{L}[\![[v, \ldots, v]]\!] = [v, \ldots, v] \qquad \text{(LIFT-LIT-LIST-TOP)}$$

$$\mathcal{L}[\![\texttt{table(t)}]\!] = \texttt{table(t)} \qquad \text{(LIFT-TABLE-TOP)}$$

Top-level iterators are handled by Rule LIFT-ITER-TOP. This rule kicks off the translation of the interesting cases: expressions that are evaluated iteratively.

$$\mathcal{L}[\![\texttt{[}\ e_1\ \texttt{|}\ x \leftarrow e_2\ \texttt{]}]\!] = \texttt{let}\ x = \mathcal{L}[\![e_2]\!]\ \texttt{in}\ \mathcal{L}[\![e_1]\!]^1_{\{x\}} \qquad \text{(LIFT-ITER-TOP)}$$

In the original iterator, head expression e_1 is evaluated in an environment with a binding for variable x. Rule LIFT-ITER-TOP creates a corresponding parallel environment by binding x to the result of the generator expression. The head expression e_1 is then lifted by function $\mathcal{L}[\![-]\!]^d_\rho$. The lifted expression $\mathcal{L}[\![e_1]\!]^1_{\{x\}}$ is placed in a parallel environment in which the lifted variable x holds all bindings for x.

ITERATIVE EXPRESSIONS For any $\mathcal{CL}$ expression that is evaluated iteratively (including nested iterators), function $\mathcal{L}[\![-]\!]^d_\rho$ derives a equivalent lifted $\mathcal{FL}$ expression. The iteration depth is denoted by d and the set ρ tracks variables that are in scope. We know that $\rho \neq \emptyset$ because at least the one variable bound by the outermost iterator is in scope, and that $d \geqslant 1$. For iterative expressions, the result of $\mathcal{L}[\![-]\!]^d_\rho$ can be summarized as follows: If e is an expression in an iteration scope at depth d, $\mathcal{L}[\![e]\!]^d_\rho$ evaluates to the result of e for all evaluations of its iteration scope.

As in Rule LIFT-ITER-TOP, lifting maintains a parallel environment for all sub-expressions. For some expression e at depth d and all variables $x \in \rho$ with $x : \tau$, $\mathcal{L}[\![e]\!]^d_\rho$ is evaluated in a parallel environment with bindings

$$x : \underbrace{[\ldots[}_{d}\tau]\ldots]$$

In this environment, x contains all bindings for x in all iterations. The lifting of variable references is easy, then:

$$\mathcal{L}[\![x]\!]^d_\rho = x \qquad \text{(LIFT-VAR)}$$

Literal expressions (v, $[v, \ldots, v]$, $\texttt{table(t)}$) are lifted by creating a copy of the constant for each iteration. This is easily achieved by replicating the constant for each element of a list that already has the required shape. We obtain such a list by choosing an arbitrary variable from the parallel environment.

$$\mathcal{L}[\![v]\!]^d_\rho = v \mathbin{\dot{\times}_d} x \quad (\text{some } x \in \rho) \qquad \text{(LIFT-LIT-ATOM)}$$

$$\mathcal{L}[\![\texttt{table(t)}]\!]^d_\rho = \texttt{table(t)} \mathbin{\divideontimes_d} x \quad (\text{some } x \in \rho) \qquad \text{(LIFT-TABLE)}$$

$$\mathcal{L}[\![[v, \ldots, v]]\!]^d_\rho = [v, \ldots, v] \mathbin{\divideontimes_d} x \quad (\text{some } x \in \rho) \qquad \text{(LIFT-LIT-LIST)}$$

The macro $v \dot{\times}_d x$ replicates the constant v for each element of list x by replacing each element of x at depth d with v. For $d = 1$, variable x denotes a flat list. In this case, replication simply amounts to an application of $\texttt{rep}\{v\}$. The case for $d > 1$ is uniformly handled by dropping the outer list structure, replicating the constant and restructuring the resulting list to the appropriate shape:

$$v \dot{\times}_1 e = \texttt{rep}\{v\}\ e$$
$$v \dot{\times}_d e = \texttt{imprint}_{d-1}\ e\ (\texttt{rep}\{v\}\ (\texttt{forget}_{d-1}\ e))$$

For list constants, we use an equivalent macro that implements replication of lists with operator $\otimes$.

$$e_1 \mathbin{\divideontimes}_1 e_2 = e_1 \otimes e_2$$
$$e_1 \mathbin{\divideontimes}_d e_2 = \texttt{imprint}_{d-1}\ e_2\ (e_1 \otimes (\texttt{forget}_{d-1}\ e_2))$$

We now consider combinators p iteratively applied to operands $e_1, \ldots, e_n$. By translating the operands, we obtain nested lists of operands of depth d. The operation p has to be performed on the elements of those lists at level d. In Rule LIFT-BUILTIN, we express this with the help of a macro $[\![-]\!]_d$:

$$\mathcal{L}[\![p\ e_1 \cdots e_n]\!]^d_\rho = [\![\ p\ \mathcal{L}[\![e_1]\!]^d_\rho\ \cdots\ \mathcal{L}[\![e_1]\!]^d_\rho\ ]\!]_d \qquad \text{(LIFT-BUILTIN)}$$

Macro $[\![\ p\ e_1 \cdots e_n\ ]\!]_d$ describes the lifted application of p to $\mathcal{FL}$ expressions $e_1, \ldots, e_n$ at depth d. By flattening the list structure with $\texttt{forget}_{d-1}$, we obtain flat lists of operands of the same length and apply the lifted combinator $p^\uparrow$:

$$[\![\ p\ e_1 \cdots e_n\ ]\!]_0 = p\ e_1 \cdots e_n$$
$$[\![\ p\ e_1 \cdots e_n\ ]\!]_1 = p^\uparrow\ e_1 \cdots e_n$$
$$[\![\ p\ e_1 \cdots e_n\ ]\!]_d = \texttt{imprint}_{d-1}\ e_1\ (p^\uparrow\ (\texttt{forget}_{d-1}\ e_1)\ \vdots\ (\texttt{forget}_{d-1}\ e_n))$$

In this definition, we restructure the flat result according to the first operand e_1. We might as well have chosen any other operand because their list shape up to depth d is the same. Note that the definition includes a case $[\![-]\!]_0$ for depth 0. For rule LIFT-BUILTIN, we know that $d \geqslant 1$. The case for $d = 0$ is required for the lifting of conditionals, however.

To lift a `let`-binding, we extend the parallel environment for expression e_2 with a binding for x. Variable x is bound to a list that contains the result of all iterative evaluations of expression e_1. This list is readily obtained by lifting e_1 itself. Rule LIFT-LET preserves the `let`-binding with both expressions lifted.

$$\mathcal{L}[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!]^d_\rho = \texttt{let}\ x = \mathcal{L}[\![e_1]\!]^d_\rho\ \texttt{in}\ \mathcal{L}[\![e_2]\!]^d_{\rho \cup \{x\}} \qquad \text{(LIFT-LET)}$$

The central lifting rule is Rule LIFT-ITER that handles nested iteration:

$$\mathcal{L}[\![[\ e_1\ |\ x \leftarrow e_2\ ]]\!]^d_\rho = \qquad \text{(LIFT-ITER)}$$
$$\texttt{let}\quad x = \mathcal{L}[\![e_2]\!]^d_\rho$$
$$\qquad [\,y = [\![\ y \otimes x\ ]\!]_d\,]_{y \in \rho, y \neq x}$$
$$\texttt{in}\ \mathcal{L}[\![e_1]\!]^{d+1}_{\rho \cup \{x\}}$$

The iterator itself is translated at depth d and opens a new iteration scope in which its head expression e_1 will be translated at depth $d+1$. First, however, the parallel environment has to be set up for e_1. A chain of `let`-bindings binds the iterator variable x and lifts all variables y that are currently in scope (*environment lifting*). We write `let` x_1 `=` e_1`, ...,` x_n `=` e_n `in` e as abbreviation for a chain of non-recursive `let`-bindings.

Lifting the generator expression e_2 determines the iteration structure for the head expression e_1. Let the generator expression e_2 be of type `[`τ`]`. Lifting e_2 results in the following type:

$$\mathcal{L}[\![e_2]\!]^d_\rho : \overbrace{\underbrace{[\ldots[}_{d}\,\tau]\ldots]}^{d+1}$$

At depth d, the result of expression $\mathcal{L}[\![e_2]\!]^d_\rho$ corresponds to iteration over lists of type `[`τ`]`. At depth $d+1$, however, it corresponds to iteration over values of type τ — this is what the iterator `[` e_1 `|` x $\leftarrow$ e_2 `]` expresses. Additionally, we lift the environment ρ to obtain the proper parallel environment for e_1. In the parallel environment of the iterator, values for all variables $y \in \rho$ reflect the iteration structure at the current depth d. Individual values for a variable y are found at list depth d. With the lifted replication operator $\otimes^{\uparrow}$, we replicate those values over the lists found at depth d of the list for the iteration variable x. At depth $d+1$, then, the result reflects an iteration over copies of values for y. As for built-in operators, we use macro $[\![-]\!]_d$ to lift $\otimes$.

Finally, we lift `if` conditionals with a standard branch-free approach as described by Prins *et al.* [PP93]. We identify those iterations in which the `then` and `else` branches, respectively, are evaluated. We split the parallel environment accordingly and lift both branches on their respective part of the parallel environment. Finally, branch results are merged to obtain the result of the conditional for all iterations.

$$\begin{aligned}
&\mathcal{L}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]^d_\rho = && \text{(LIFT-COND)}\\
&\quad [\![\texttt{combine bs}\\
&\qquad\qquad (\texttt{let } [x = [\![\texttt{restrict } [\![\langle x, \texttt{bs}\rangle]\!]_d]\!]_{d-1}]_{x\in\rho}\\
&\qquad\qquad\ \texttt{in } \mathcal{L}[\![e_2]\!]^d_\rho\ )\\
&\qquad\qquad (\texttt{let } [x = [\![\texttt{restrict } [\![\langle x, [\![\texttt{not bs}]\!]_d\rangle]\!]_d]\!]_{d-1}]_{x\in\rho}\\
&\qquad\qquad\ \texttt{in } \mathcal{L}[\![e_3]\!]^d_\rho\ )\ [\![]\!]_{d-1}\\
&\quad \text{where } \texttt{bs} = \mathcal{L}[\![e_1]\!]^d_\rho
\end{aligned}$$

Note that `restrict` and `combine` are applied with $[\![-]\!]_{d-1}$. At depth d, we have values for individual iterations. To filter and combine them, however, we need lists of of those values and therefore work at depth $d-1$. This is the reason why $\mathcal{FL}$ includes combinators `restrict` and `combine` in non-parallel form: if a conditional is translated at depth $d = 1$, we require `restrict` and `combine` on single arguments.

4.2.3 *Example*

In Section 3.2 we have eliminated iterators by replicating constants, environment lifting and lifted combinators. Function $\mathcal{L}[\![-]\!]$ is an implementation of these concepts. Indeed, we can easily verify that applying $\mathcal{L}[\![-]\!]$ to our

running example Query Q2 derives the iterator-free $\mathcal{FL}$ expression in Figure 21b. Only minor syntactical differences have to be observed: we write $\mathtt{p}^{\uparrow}$ instead of $\mathtt{p}^{1}$ and $\llbracket\, \mathtt{p} \,\rrbracket_{\mathtt{d}}$ instead of $\mathtt{p}^{\mathtt{d}}$.

Function $\mathcal{L}[\![-]\!]$ proceeds mechanically on the syntactical structure of a $\mathcal{CL}_{\mathtt{d}}$ expression. As an example for the handling of the individual $\mathcal{CL}$ constructs, we trace the translation of Query Q5 (Section 4.1). Query Q5 computes for each order the list of matching line items based on an `if` conditional. All translation steps are shown in Figure 28.

The resulting $\mathcal{FL}$ query sets up parallel environments containing bindings for variable `o` for the outer iteration scope and `l` and `o` for the inner iteration scope. Data replication ($\mathtt{ls} \mathbin{\divideontimes_1} \mathtt{os}$) and environment lifting ($\llbracket\, \mathtt{os} \otimes \mathtt{l} \,\rrbracket_1$) construct nested lists of the appropriate shape. Data-parallel operations are used on these nested lists: $\llbracket\, \llbracket\, \mathtt{l.ok} \,\rrbracket_2 = \llbracket\, \mathtt{o.ok} \,\rrbracket_2 \,\rrbracket_2$ derives the data-parallel variant of the predicate `l.ok == o.ok`.

Let us take a closer look at the sub-expression derived with rule LIFT-COND that evaluates the conditional in the innermost iterator in a data-parallel fashion. We assume that `os` is a list of two order records $[\mathtt{o}_1, \mathtt{o}_2]$ and `ls` is a list of four line item records $[\mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3, \mathtt{l}_4]$. If $\mathtt{o}_1$ and $\mathtt{o}_2$ match the line items $\mathtt{l}_1$, $\mathtt{l}_2$ and $\mathtt{l}_4$, and $\mathtt{l}_3$, respectively, we have the following intermediate lists in the innermost scope:

$$
\begin{array}{lcl}
\mathtt{l} & \equiv & [[\ \mathtt{l}_1\ ,\ \mathtt{l}_2\ ,\ \mathtt{l}_3\ ,\ \mathtt{l}_4\ ]],[[\ \mathtt{l}_1\ ,\ \mathtt{l}_2\ ,\ \mathtt{l}_3\ ,\ \mathtt{l}_4\ ]] \\
\mathtt{o} & \equiv & [[\ \mathtt{o}_1\ ,\ \mathtt{o}_1\ ,\ \mathtt{o}_1\ ,\ \mathtt{o}_1\ ]],[[\ \mathtt{o}_2\ ,\ \mathtt{o}_2\ ,\ \mathtt{o}_2\ ,\ \mathtt{o}_2\ ]] \\
\mathtt{bs} & \equiv & [[\,\mathtt{True}\,,\mathtt{True}\,,\mathtt{False}\,,\mathtt{True}\,]],[[\,\mathtt{False}\,,\mathtt{False}\,,\mathtt{True}\,,\mathtt{False}\,]]
\end{array}
$$

The result `bs` of the predicate dictates the assignment of individual values of `l` and `o` to the respective conditional branches. With $\mathtt{restrict}^{\uparrow}$, we derive the following bindings for `l` in the restricted environments for both branches:

$$
\begin{array}{ll}
\mathtt{l} \equiv [[\mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_4], [\mathtt{l}_3]] & (\texttt{then-branch}) \\
\mathtt{l} \equiv [[\mathtt{l}_3], [\mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_4]] & (\texttt{else-branch})
\end{array}
$$

Branch expressions are evaluated in parallel in the restricted environments to obtain the collective branch results.

$$
\begin{array}{l}
\llbracket\, \mathtt{sng\ l} \,\rrbracket_2 \equiv [[[\mathtt{l}_1], [\mathtt{l}_2], [\mathtt{l}_4]], [[\mathtt{l}_3]]] \\
\quad [] \mathbin{\divideontimes_2} \mathtt{l} \equiv [[[]], [[], [], []]]
\end{array}
$$

We merge branch results with $\mathtt{combine}^{\uparrow}$. Finally, $\mathtt{concat}^{\uparrow}$ flattens the inner lists, resulting in the overall query result.

$$
\begin{array}{l}
\llbracket\, \mathtt{combine\ ...} \,\rrbracket_1 \equiv [[[\mathtt{l}_1], [\mathtt{l}_2], [], [\mathtt{l}_4]], [[], [], [\mathtt{l}_3], []]] \\
\llbracket\, \mathtt{concat\ ...} \,\rrbracket_1 \equiv [[\mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_4], [\mathtt{l}_3]]
\end{array}
$$

4.2.4 *Indexed Semantics of $\mathcal{FL}$*

We define the semantics of $\mathcal{FL}$ based on indexed lists in terms of $\mathcal{ML}$. We focus on a couple of $\mathcal{FL}$ combinators to discuss the essential cases and list the remaining rules in Appendix A.

$$
\begin{array}{l}
\mathcal{L}[\![\ \texttt{[concat [if l.ok = o.ok then sng l else [] | l} \leftarrow \texttt{ls]} \\
\quad \texttt{o} \leftarrow \texttt{os]}\!] \\
\equiv \quad \{\ \textsc{lift-iter-top}\ \} \\
\quad \texttt{let o = os} \\
\quad \texttt{in } \mathcal{L}[\![\texttt{concat [if l.ok = o.ok then sng l else [] | l} \leftarrow \texttt{ls]}]\!]^1_{\{\texttt{o}\}} \\
\equiv \quad \{\ \textsc{lift-table-top}, \textsc{lift-builtin}\ \} \\
\quad \texttt{let o = os} \\
\quad \texttt{in } \lceil\!\lceil \texttt{concat } \mathcal{L}[\![\texttt{[if l.ok = o.ok then sng l else [] | l} \leftarrow \texttt{ls]}]\!]^1_{\{\texttt{o}\}} \rceil\!\rceil_1 \\
\equiv \quad \{\ \textsc{lift-iter}\ \} \\
\quad \texttt{let o = os} \\
\quad \texttt{in } \lceil\!\lceil \texttt{concat (let l = } \mathcal{L}[\![\texttt{ls}]\!]^1_{\{\texttt{o}\}} \texttt{, o = } \lceil\!\lceil \texttt{o} \otimes \texttt{l} \rceil\!\rceil_1 \\
\qquad\qquad \texttt{in } \mathcal{L}[\![\texttt{if l.ok = o.ok then sng l else []}]\!]^2_{\{\texttt{l,o}\}} \texttt{)} \rceil\!\rceil_1 \\
\equiv \quad \{\ \textsc{lift-table}\ \} \\
\quad \texttt{let o = os} \\
\quad \texttt{in } \lceil\!\lceil \texttt{concat (let l = ls} \ltimes_1 \texttt{os, o = } \lceil\!\lceil \texttt{os} \otimes \texttt{l} \rceil\!\rceil_1 \\
\qquad\qquad \texttt{in } \mathcal{L}[\![\texttt{if l.ok = o.ok then sng l else []}]\!]^2_{\{\texttt{l,o}\}} \texttt{)} \rceil\!\rceil_1 \\
\equiv \quad \{\ \text{inline variable o}\ \} \\
\quad \lceil\!\lceil \texttt{concat (let l = ls} \ltimes_1 \texttt{os, o = } \lceil\!\lceil \texttt{os} \otimes \texttt{l} \rceil\!\rceil_1 \\
\qquad\qquad \texttt{in } \mathcal{L}[\![\texttt{if l.ok = o.ok then sng l else []}]\!]^2_{\{\texttt{l,o}\}} \texttt{)} \rceil\!\rceil_1 \\
\equiv \quad \{\ \textsc{lift-cond}\ \} \\
\quad \lceil\!\lceil \texttt{concat (let l = ls} \ltimes_1 \texttt{os, o = } \lceil\!\lceil \texttt{os} \otimes \texttt{l} \rceil\!\rceil_1 \\
\qquad\qquad \texttt{in let bs = } \mathcal{L}[\![\texttt{l.ok = o.ok}]\!]^2_{\{\texttt{l,o}\}} \\
\qquad\qquad\quad \texttt{in } \lceil\!\lceil \texttt{combine} \\
\qquad\qquad\qquad\quad \texttt{bs} \\
\qquad\qquad\qquad\quad \texttt{(let l = } \lceil\!\lceil \texttt{restrict } \lceil\!\lceil \langle \texttt{l, bs} \rangle \rceil\!\rceil_2 \rceil\!\rceil_1 \\
\qquad\qquad\qquad\qquad \texttt{in } \mathcal{L}[\![\texttt{sng l}]\!]^2_{\{\texttt{l,o}\}} \texttt{)} \\
\qquad\qquad\qquad\quad \texttt{(let l = } \lceil\!\lceil \texttt{restrict } \lceil\!\lceil \langle \texttt{l, } \lceil\!\lceil \texttt{not bs} \rceil\!\rceil_2 \rangle \rceil\!\rceil_2 \rceil\!\rceil_1 \\
\qquad\qquad\qquad\qquad \texttt{in } \mathcal{L}[\![\texttt{[]}]\!]^2_{\{\texttt{l,o}\}} \texttt{)} \rceil\!\rceil_1 \texttt{)} \rceil\!\rceil_1 \\
\equiv \quad \{\ \textsc{lift-builtin}, \textsc{lift-var}, \textsc{lift-lit-list}\ \} \\
\quad \lceil\!\lceil \texttt{concat (let l = ls} \ltimes_1 \texttt{os, o = } \lceil\!\lceil \texttt{os} \otimes \texttt{l} \rceil\!\rceil_1 \\
\qquad\qquad \texttt{in let bs = } \lceil\!\lceil \lceil\!\lceil \texttt{l.ok} \rceil\!\rceil_2 = \lceil\!\lceil \texttt{o.ok} \rceil\!\rceil_2 \rceil\!\rceil_2 \\
\qquad\qquad\quad \texttt{in } \lceil\!\lceil \texttt{combine} \\
\qquad\qquad\qquad\quad \texttt{bs} \\
\qquad\qquad\qquad\quad \texttt{(let l = } \lceil\!\lceil \texttt{restrict } \lceil\!\lceil \langle \texttt{l, bs} \rangle \rceil\!\rceil_2 \rceil\!\rceil_1 \\
\qquad\qquad\qquad\qquad \texttt{in } \lceil\!\lceil \texttt{sng l} \rceil\!\rceil_2 \texttt{)} \\
\qquad\qquad\qquad\quad \texttt{(let l = } \lceil\!\lceil \texttt{restrict } \lceil\!\lceil \langle \texttt{l, } \lceil\!\lceil \texttt{not bs} \rceil\!\rceil_2 \rangle \rceil\!\rceil_2 \rceil\!\rceil_1 \\
\qquad\qquad\qquad\qquad \texttt{in [] } \ltimes_2 \texttt{ l) } \rceil\!\rceil_1 \texttt{) } \rceil\!\rceil_1
\end{array}
$$

Figure 28: Lifting of Query Q5.

Indexed lists are originally derived from literal lists and table references as in $\mathcal{CL}_d$.

$$\mathcal{F}[\![[v_1, \ldots, v_n]]\!]_\rho = [\langle k = 1, p = [\![v_1]\!]\rangle, \ldots, \langle k = n, p = [\![v_n]\!]\rangle] \quad (\mathcal{FL}\text{-LIST})$$

$$\mathcal{F}[\![\texttt{table(t)}]\!]_\rho = [\ \langle k = \mathrm{pk}_t(x), p = x\rangle \mid x \leftarrow \mathsf{sortWith}\ (\lambda x.\, \mathrm{pk}_t(x))\ [\![t]\!]\ ] \quad (\mathcal{FL}\text{-TABLE})$$

Crucial are the shape operators `forget` and `imprint`. First, `forget` temporarily flattens one nesting level of a nested list. For the flattened list, it constructs unique indexes as pairs of the outer index (xs.k) and the inner index (x.k). To reconstruct nesting, `imprint` exploits this specific structure of indexes.

$$\mathcal{F}[\![\texttt{forget}\ e]\!]_\rho = [\ \langle k = \langle xs.k, x.k\rangle, p = x.p\rangle \mid xs \leftarrow \mathcal{F}[\![e]\!]_\rho, x \leftarrow xs.p\ ] \quad (\mathcal{FL}\text{-FORGET})$$

$$\mathcal{F}[\![\texttt{imprint}\ e_1\ e_2]\!]_\rho = [\ \langle k = x.k, p = [\ \langle k = y.k.2, p = y.p\rangle \mid y \leftarrow \mathcal{F}[\![e_2]\!]_\rho, x.k = y.k.1\ ]\rangle \mid x \leftarrow \mathcal{F}[\![e_1]\!]_\rho\ ] \quad (\mathcal{FL}\text{-IMPRINT})$$

Note that the semantics of `forget` matches that of `concat` defined in Equation ($\mathcal{FL}$-CONCAT) (Appendix A). It is nevertheless necessary to distinguish the two operators. Once we lower to a flat representation, their implementation will differ.

We only define `imprint` and `forget` although lifting generates the more general $\texttt{imprint}_d$ and $\texttt{forget}_d$ with an arbitrary depth d. However, it is easy to see that these are related as follows [KS96]:

$$\texttt{forget}_d\ e = \texttt{forget}\ (\texttt{forget}_{d-1}\ e)$$
$$\texttt{forget}_1\ e = \texttt{forget}\ e$$

$$\texttt{imprint}_d\ e_1\ e_2 = \texttt{imprint}\ e_1\ (\texttt{imprint}_{d-1}\ (\texttt{forget}\ e_1)\ e_2)$$
$$\texttt{imprint}_1\ e_1\ e_2 = \texttt{imprint}\ e_1\ e_2$$

For $\texttt{forget}_d$ we strip nesting layers one by one. Inversely, for $\texttt{imprint}_d$ we apply nesting layers one by one. We note that the index-based interpretation of the reshaping operators bears close resemblance to the nest and unnest operators of nested relational algebra.

The replication combinators simply repeat the respective value:

$$\mathcal{F}[\![e_1 \otimes e_2]\!]_\rho = [\ \langle k = x.k, p = \mathcal{F}[\![e_1]\!]_\rho\rangle \mid x \leftarrow \mathcal{F}[\![e_2]\!]_\rho\ ] \quad (\mathcal{FL}\text{-DIST})$$

$$\mathcal{F}[\![\texttt{rep\{v\}}\ e]\!]_\rho = [\ \langle k = x.k, p = v\rangle \mid x \leftarrow \mathcal{F}[\![e]\!]_\rho\ ] \quad (\mathcal{FL}\text{-DIST-BASE})$$

Lifted combinators are the remaining interesting category of $\mathcal{FL}$ combinators. Lifted combinators with a single (lifted) argument are straightforward to define. The operation is applied to each element of the argument list,

regardless whether it is a scalar or list operation. For lifted list combinators, indexes of inner lists are maintained as in Section 4.1.2.

$$\mathcal{F}[\![e.\ell^{\uparrow}]\!]_\rho = [\ \langle k = x.k, p = x.p.\ell\rangle \mid x \leftarrow \mathcal{F}[\![e]\!]_\rho\ ] \qquad (\mathcal{FL}\text{-RECORD-SEL-LIFT})$$

$$\begin{aligned}&\mathcal{F}[\![\texttt{number}^{\uparrow}\ e]\!]_\rho = \\ &\quad [\ \langle k = xs.k, p = [\ \langle k = x.1.k, p = \langle x.1.p, x.2\rangle\rangle \\ &\qquad\qquad \mid x \leftarrow \texttt{enum}\ xs.p\ ]\rangle \\ &\quad \mid xs \leftarrow \mathcal{F}[\![e]\!]_\rho\ ]\end{aligned} \qquad (\mathcal{FL}\text{-NUMBER-LIFT})$$

$$\begin{aligned}&\mathcal{F}[\![\texttt{restrict}^{\uparrow}\ e]\!]_\rho = \\ &\quad [\ \langle k = xs.k, p = [\ \langle k = x.k, p = x.p.1\rangle \mid x \leftarrow xs.p, x.p.2\ ]\rangle \\ &\quad \mid xs \leftarrow \mathcal{F}[\![e]\!]_\rho]\end{aligned}$$

$$(\mathcal{FL}\text{-RESTRICT-LIFT})$$

Note that shape and indexes of the outer list are preserved. Only the *elements* of the outer lists are modified.

More interesting are lifted combinators with more than one argument. Consider the following example of the lifted $+^{\uparrow}$ combinator:

```
+↑ [2, 5, 3]
   [7, 4, 6]
```

To evaluate $+^{\uparrow}$, corresponding elements of both arguments need to be aligned. In the flattening transformation, this alignment is usually described in terms of list positions [PP93] or explicit order-aware combinators like `zip` [KS96]. This, however, would tie us to an ordered representation of lists. Instead, we use indexes to align corresponding elements:

```
+↑ [⟨k=1, p=2⟩, ⟨k=2, p=2⟩, ⟨k=3, p=2⟩]
   [⟨k=1, p=7⟩, ⟨k=2, p=4⟩, ⟨k=3, p=6⟩]
```

Lifted combinators are applied to argument expressions that are lifted in the same iteration scope. Consequentially, all argument expressions of a lifted combinator have the same outer list shape and the same indexes. This allows us to align elements from corresponding iterations by joining arguments on their indexes.

$$\begin{aligned}\mathcal{F}[\![c(e_1, e_2)^{\uparrow}]\!]_\rho = [\ &\langle k = x.k, p = [\![c]\!](x.p, y.p)\rangle \\ &\mid x \leftarrow \mathcal{F}[\![e_1]\!]_\rho, y \leftarrow \mathcal{F}[\![e_2]\!]_\rho, x.k = y.k\ ]\end{aligned}$$

$$(\mathcal{FL}\text{-BASE-OP-LIFT})$$

$$\begin{aligned}\mathcal{F}[\![e_1 \otimes^{\uparrow} e_2]\!]_\rho = [\ &\langle k = x.k, p = [\ \langle k = y.k, p = x.p\rangle \mid y \leftarrow ys.p\ ]\rangle \\ &\mid x \leftarrow \mathcal{F}[\![e_1]\!]_\rho, ys \leftarrow \mathcal{F}[\![e_2]\!]_\rho, x.k = ys.k\ ]\end{aligned}$$

$$(\mathcal{FL}\text{-DIST-LIFT})$$

4.3 FLATTENING COLLECTIONS: THE SEGMENT VECTOR MODEL

Lifting eliminates nested iteration from queries by translating into $\mathcal{FL}$. This language is defined on lists, *i.e.* ordered and potentially nested collections. Ultimately, though, we want to target query engines centered around flat collections — in particular, flat unordered multisets. In this section, we introduce *segment vectors* as a flat, ordered representation of nested lists that mediates between those worlds. We translate $\mathcal{FL}$ expressions into a language of operators on flat vectors.

Although we mainly target unordered collections, we introduce an ordered flat intermediate representation first and translate to an unordered representation in a subsequent step (Chapter 6). This design choice is beneficial for two reasons: First, we obtain simple translation steps that handle only single, well-defined aspects. Second, ordered vectors as an intermediate representation enable us to generate code for order-aware backends as well.

Terminology for collection types in literature on the flattening transformation is non-uniform and depends on the respective target platform and language. Blelloch [Ble90] mostly uses the term *vector*, while other authors use *(parallel) array* [CK00; Pey+08; Kel99] and *sequence* [PP93; PP95]. In the following, we use the term *vector* for flat, homogeneous, ordered sequences of scalar values. Our translation does *not* depend on positional access to vectors. When defining the semantics of our vector language, we interpret vectors as meta-level lists. Accordingly, we use list notation in examples.

4.3.1 *Segment Vectors*

Blelloch and Sabot [BS89] describe a flat representation of nested collections in terms of flat *segment vectors*. In this non-parametric model [Pey+08], the representation of collections depends on the type of elements. Collections of atomic values are represented as flat vectors. Collections whose elements are not scalar (*e.g.* lists of types `[[Int]]` and `[⟨Int,[Int]⟩]`), on the other hand, are represented by separating structure from content. Atomic values that make up the actual content of the complex value are stored in flat vectors of atomic values, whereas their structure in the complex value is recorded separately.

In Blelloch's representation, complex values are flattened as follows:

- If the element type of a list is itself a list, (*e.g.* type `[[Int]]`), the flat representation is a pair of two vectors. A *data vector* stores the elements of all inner lists. A separate *segment descriptor* vector describes a partitioning of the data vector into segments and encodes the structure of the outer list. Usually, the segment descriptor records the lengths of the segments as in the following example of a list of type `[[Int]]`:

  ```
  [[10, 20, 30], [], [70, 80]]
  ⇒
  ⟨[3, 0, 2], [10,20,30,70,80]⟩
  ```

  Note that the empty inner list has no explicit representation in the data vector. It is encoded as an empty segment of length 0.

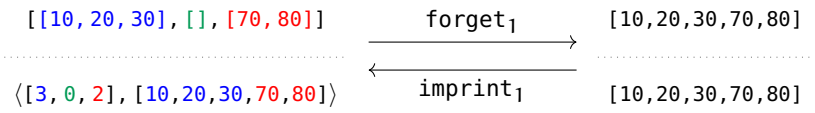
  This representation extends to arbitrary nesting depths. For instance, a list of type `[[[Int]]]` is represented by two segment descriptors and one data vector.

- Lists of records are decomposed into records of vectors that store the content of individual record fields. For example, a list of type `[⟨Int,[Int]⟩]` is represented as follows:

  ```
  [⟨100, [10, 20, 30]⟩, ⟨200, []⟩, ⟨300, [70, 80]⟩]
  ⇒
  ⟨[100, 200, 300], ⟨[3, 0, 2], [10,20,30,70,80]⟩⟩
  ```

The three pair elements of the original outer list are represented as the three elements of the data vector `[100, 200, 300]` and the segment descriptor `[3, 0, 2]`.

On this representation, the shape operators `forget` and `imprint` have no runtime cost and just rearrange vectors. Data vectors represent flat lists of atomic values. Simply ignoring associated segment descriptors temporarily flattens lists:



Most subsequent work on the flattening transformation uses essentially the same data representation with length-based segment descriptors and flat atomic vectors [Ble+94; Pey+08; PP93; Les05; Ber+13; PP95; Kel99]. Chakravarty and Keller [CK00] extend the flat representation to general sum and product types, including recursive types. Next to length-based segment descriptors, Blelloch [Ble90] discusses other possible encodings of the segment information (storing flags that describe a change of the segment, and positions of the start of each segment, respectively). However, with one exception [BR12], only length-based segment descriptors are used in the literature. Lippmeier *et al.* [Lip+12] extend segment descriptors to allow sharing of segments (see also Chapter 7).

4.3.1.1 *Length-Based Vectors and Query Processing*

Although used prevalently in the literature, Blelloch's flat representation is not a good fit for all conceivable target platforms. For example, Bergstrom *et al.* [BR12] point out that length-based segment descriptors do not allow an efficient implementation of some of the required vector operations on GPUs. Directly adapting the length-based vector model is not an option in our setting of query processing either.

Length-based segment descriptors lead to code that frequently recomputes segment lengths and accesses vector elements by position:

- In queries, filtering collections is ubiquitous. In $\mathcal{FL}$, `restrict`$^{\uparrow}$ expresses filtering of lists:

  `restrict`$^{\uparrow}$ `[[⟨1, True⟩, ⟨2, False⟩, ⟨3, True⟩], [⟨4, True⟩, ⟨5, False⟩]]`

  With length-based segment descriptors, we obtain the following flat representation for the argument list, composed of one segment descriptor and two data vectors:

  `⟨[3,2], ⟨[1,2,3,4,5], [True,False,True,True,False]⟩⟩`

  Filtering itself is performed on the data vector. To keep the representation consistent however, we have to update segment lengths to account for elements that have been removed. In general, lifted list combinators that change the shape of lists (*e.g.* `distinct`$^{\uparrow}$, `restrict`$^{\uparrow}$) are *non-local*: they have to recompute and update segment lengths next to the actual data vector.

- Data-parallel scalar operators like $+^{\uparrow}$ work on multiple lists of arguments that are encoded in multiple vectors. If data vectors are simple

vectors of atomic values, corresponding elements of multiple vectors can only be identified by their position. Vectors have to be aligned positionally. When lowering vectors to unordered multisets, this would force us to encode actual list positions, not just a relative order of elements.

- $\mathcal{FL}$ is centered around a number of lifted collection primitives (*e.g.* $\texttt{sort}^{\uparrow}$, $\texttt{group}^{\uparrow}$). Applying $\texttt{sort}^{\uparrow}$, for example, sorts each inner list independently:

  $$\texttt{sort}^{\uparrow}\ [[\langle 10, 2\rangle, \langle 20, 1\rangle, \langle 30, 3\rangle], [\langle 40, 6\rangle, \langle 50, 4\rangle]]$$

  A vector operator that implements the $\texttt{sort}^{\uparrow}$ combinator needs to sort individual segments of the data vector. With length-based vectors, however, segment boundaries can only be determined from the segment descriptor. An implementation of $\texttt{sort}^{\uparrow}$ has to align data vectors and segment descriptors to compute segment boundaries, an operation that is inherently positional. Furthermore, it introduces data dependencies between segment descriptors and data vectors.

Computation of lengths and positional access are cheap if vectors are mapped to physical arrays with constant-time positional element access. In our setting, however, these operations are not cheap at all. Most query engines use pipelining and avoid the materialization of intermediate results as long as possible [Gra93]. Single tuples or blocks of tuples [BZN05] are streamed through pipelines of query operators. Computing lengths breaks the pipeline and enforces the materialization of the input: the length can only be computed once the complete input is available. Furthermore, if the query engine offers only unordered collections, positional access requires sorting and forces materialization of the input as well. Generating backend code that implements length-based vectors would effectively disable a query engine's ability to stream data.

Lists records are represented as multiple vectors of atomic values. In a main-memory setting, these vectors can be implemented as unboxed arrays with data locality properties that play well with the CPU cache hierarchy. In query processing, similar decomposed storage models are well-known. Copeland and Khoshafian [CK85] propose a decomposition of n-ary relations into single columns. A number of *column-store* query engines (*e.g.* MonetDB [MKB09]) implements this scheme. Subscribing to the decomposed representation of records in our intermediate representation would be premature, however. A substantial number of relevant database systems do not implement a columnar processing model. Even systems that are based on a columnar storage model internally typically do not expose it but offer a higher-level interface with n-ary relations. For example, the physical algebra implemented by the X100 engine [BZN05] works on relations with n-ary tuples. The sole exception is the MonetDB system [MKB09] with its columnar language MAL. A non-decomposed vector model with proper records does not prohibit backends that generate optimized code for column stores and exploit their internal representation. Turning vectors of records into records of vectors is possible as a separate step after shredding.

4.3.1.2 *Index-Based Vectors*

As outlined above, the length-based encoding of segments commonly used in work on the flattening transformation is not a good fit for query engines.

However, the flattening transformation does not rely on this particular representation. In Section 4.2.1, we have defined the semantics of $\mathcal{FL}$ constructs based on lists explicitly annotated with scalar *index* values that express element identity. Here, we define an *index-based* representation of segment vectors along the lines of flat representations discussed in Chapter 2.

We write our example of type `[[Int]]` as an indexed list with indexes of type `Int`:

$$
\begin{aligned}
&[[10, 20, 30], [], [70, 80]] \\
\Rightarrow\ &[\langle k = 1, p = [\langle k = 1, p = 10\rangle, \langle k = 2, p = 20\rangle, \langle k = 3, p = 30\rangle]\rangle, \\
&\ \langle k = 2, p = []\rangle, \\
&\ \langle k = 3, p = [\langle k = 1, p = 70\rangle, \langle k = 2, p = 80\rangle]\rangle]
\end{aligned}
$$

Index values of the outer list uniquely identify the inner lists and the respective segments. We include these outer index values in the inner vector to mark segments explicitly. We obtain the following flat representation with two vectors V_o and V_i:

$$
V_o = \begin{bmatrix} \langle s = \langle\rangle, k = 1, p = \langle\rangle\rangle, \\ \langle s = \langle\rangle, k = 2, p = \langle\rangle\rangle, \\ \langle s = \langle\rangle, k = 3, p = \langle\rangle\rangle \end{bmatrix}
\qquad
V_i = \begin{bmatrix} \langle s = 1, k = \langle 1, 1\rangle, p = 10\rangle, \\ \langle s = 1, k = \langle 1, 2\rangle, p = 20\rangle, \\ \langle s = 1, k = \langle 1, 3\rangle, p = 30\rangle, \\ \langle s = 3, k = \langle 3, 1\rangle, p = 70\rangle, \\ \langle s = 3, k = \langle 3, 2\rangle, p = 80\rangle \end{bmatrix}
$$

Vector elements are scalar records of type $\langle s : \alpha, k : \beta, p : \gamma\rangle$, consisting of an *outer index* of type α, an *inner index* of type β and a *payload* of type γ. The outer index or *segment identifier* denotes the segment to which a vector element belongs and relates an inner vector to the corresponding outer vector. The inner index uniquely identifies vector elements. The indexes of the original individual inner lists do not uniquely identify elements of the inner vector V_i. In this example, we obtain unique inner indexes for V_i by forming pairs of the indexes of the original outer and inner lists. The payload stores the actual content of a list element. Since the payload has to be scalar, we replace all nested list values with the placeholder $\langle\rangle$.

Note that elements of the outer vector include an outer index of type $\langle\rangle$, even though the outer vector only stores the elements of the outer list. This leads to a uniform representation: all vector elements are part of a segment. Elements of the outermost list are part of the *unit segment* $\langle\rangle$.

As in Blelloch's length-based encoding, empty (inner) lists are represented by absence of a corresponding segment. In our example, the inner index 2 of the outer vector V_o does not occur as an outer index in the inner vector V_i. In general, empty inner lists can be identified by computing the set difference of outer and inner indexes of the corresponding vectors. Empty top-level lists are represented as an empty vector.

In the remainder of this thesis, we find it convenient to write $(\_, \_, \_)$ both for the record type constructor $\langle s : \_, k : \_, p : \_\rangle$ and for the record constructor $\langle s = \_, k = \_, p = \_\rangle$.

SHREDDED PACKAGES Index-based vectors store list elements, with inner list values replaced with $\langle\rangle$. This encoding results in a direct correspon-

dence between lists and vectors. Consider the following indexed list of type $[\langle \texttt{Int}, [\texttt{Int}]\rangle]$:

$$
\begin{aligned}
&[\langle 100, [10, 20, 30]\rangle, \langle 200, []\rangle, \langle 300, [70, 80]\rangle] \\
\Rightarrow\ &[\langle k=1, p=\langle 100, [\langle k=1, p=10\rangle, \langle k=2, p=20\rangle, \langle k=3, p=30\rangle]\rangle\rangle, \\
&\ \langle k=2, p=\langle 200, []\rangle\rangle, \\
&\ \langle k=3, p=\langle 300, [\langle k=1, p=70\rangle, \langle k=2, p=80\rangle]\rangle\rangle]
\end{aligned}
$$

This list is encoded with two vectors V_o' and V_i':

$$
V_o' = \begin{bmatrix} (\langle\rangle, 1, \langle 100, \langle\rangle\rangle) \\ (\langle\rangle, 2, \langle 200, \langle\rangle\rangle) \\ (\langle\rangle, 3, \langle 300, \langle\rangle\rangle) \end{bmatrix} \qquad V_i' = \begin{bmatrix} (1, \langle 1, 1\rangle, 10) \\ (1, \langle 1, 2\rangle, 20) \\ (1, \langle 1, 3\rangle, 30) \\ (3, \langle 3, 1\rangle, 70) \\ (3, \langle 3, 2\rangle, 80) \end{bmatrix}
$$

The segment vector representations of our examples of types $[[\texttt{Int}]]$ and $[\langle \texttt{Int}, [\texttt{Int}]\rangle]$ are mostly identical and differ only in the payload of the outer vectors V_o and V_o'. The only difference between the types occurs in the scalar component of the outer element type. Their list nesting structure, on the other hand, is the same. Only the list nesting structure dictates the structure of the vector representation.

List types and vectors are related directly: each list type constructor $[\_]$ corresponds to one vector. Following Cheney *et al.* [CLW14a], we organize vectors into *shredded packages* (short: *package*). A shredded package is a type ρ in which each list type constructor is annotated with a segment vector:

$$\rho ::= \pi \mid [\rho]^V \mid \langle \ell : \rho, \ldots, \ell : \rho \rangle$$

A shredded package annotated with segment vectors represents a list value. For example, the shredded packages for the example lists in this section are $[[\texttt{Int}]^{V_i}]^{V_o}$ and $[\langle \texttt{Int}, [\texttt{Int}]^{V_i'}\rangle]^{V_o'}$.

Recall from Section 4.1 that we only consider list-typed $\mathcal{CL}$ expressions for *Query Flattening*. Accordingly, the result of any type-correct $\mathcal{CL}$ expression can be represented as a shredded package with the corresponding vectors. The lowering of $\mathcal{FL}$ expressions to vector operators (Section 4.3.3) will reflect this correspondence.

SHREDDING LISTS We now describe the representation of nested lists by segment vectors precisely. We define functions that convert between both representations. Function $\mathit{shred}([\langle k:\delta, p:\tau\rangle], xs)$ shreds an indexed list value $xs : [\langle k:\delta, p:\tau\rangle]$ into a package. Function $\mathit{stitch}([\rho]^V)$ implements the inverse transformation and stitches a nested list value from a shredded package $[\rho]^V$.

Function $\mathit{shred}(-,-)$ shreds an indexed list value into the different levels of list nesting. It produces a shredded package in which each list type constructor carries the vector that assembles the elements of all list elements at the respective nesting level. From the type alone, we can derive an initial shredded package with empty vectors:

$$
\begin{aligned}
\mathit{init}([\tau]) &= [\,\mathit{init}(\tau)\,]^{[]} \\
\mathit{init}(\langle \ell_i : \tau_i \rangle_{i=1}^n) &= \langle \ell_i : \mathit{init}(\tau_i) \rangle_{i=1}^n \\
\mathit{init}(\pi) &= \pi
\end{aligned}
$$

From a list element value, function $payload(-)$ derives the corresponding vector payload value by replacing any inner lists with $\langle\rangle$.

$$\begin{aligned} payload(\langle \ell_i = v_i \rangle_{i=1}^n) &= \langle \ell_i = payload(v_i) \rangle_{i=1}^n \\ payload(v) &= v \\ payload([\_]) &= \langle\rangle \end{aligned}$$

Function $shred_s(\tau, v)$ traverses a value v and its type τ while tracking the current index value s. Once an inner list value is encountered, it derives a vector V that contains only the segment for the current inner list value. The inner index of vector V is composed from the original index of the list and the current outer index s. At the same time, the annotated version of the list element type is derived by recursively shredding the list element values. Here, the tracked index value is updated to the composition of list index and s. By shredding the individual list elements, we obtain a list of packages that have the same structure but differ in their annotations. We merge these packages into one to obtain the package ρ that encodes the list elements.

$$\begin{aligned} shred([\langle \mathtt{k}:\delta, \mathtt{p}:\tau \rangle], xs) &= shred_{\langle\rangle}([\langle \mathtt{k}:\delta, \mathtt{p}:\tau \rangle], xs) \\ shred_s(\pi, v) &= \pi \\ shred_s(\langle \ell_i : \tau_i \rangle_{i=1}^n, \langle \ell_i = v_i \rangle_{i=1}^n) &= \langle \ell_i : shred_s(\tau_i, v_i) \rangle_{i=1}^n \\ shred_s([\langle \mathtt{k}:\delta, \mathtt{p}:\tau \rangle], xs) &= [\rho]^V \\ \text{where } \rho &= merge([\ shred_{\langle s, x.k \rangle}(\tau, x.\mathtt{p}) \mid x \leftarrow xs\ ], \tau) \\ V &= [\ (s, \langle s, x.k \rangle, payload(x.\mathtt{p})) \mid x \leftarrow xs\ ] \end{aligned}$$

Note that shredding of lists preserves the order of list elements.

Packages are merged pairwise by merging the vector annotations on list type constructors:

$$\begin{aligned} mergePkg([\rho]^V, [\rho']^{V'}) &= [\ mergePkg(\rho, \rho')]^{V +\!\!+ V'} \\ mergePkg(\langle \ell_i : \rho_i \rangle_{i=1}^n, \langle \ell_i : \rho'_i \rangle_{i=1}^n) &= \langle \ell_i : mergePkg(\rho_i, \rho'_i) \rangle \\ mergePkg(\pi, \pi) &= \pi \end{aligned}$$

$$\begin{aligned} merge([], \tau) &= init(\tau) \\ merge([\rho_1, \ldots, \rho_n], \tau) &= mergePkg(\rho_1, \ldots mergePkg(\rho_{n-1}, \tau_n)) \end{aligned}$$

Merging packages as specified results in vectors with *dense* segments: elements of one particular segment are stored consecutively.

Having defined the shredding of lists, we can now derive the vector representation of our examples from above systematically. To illustrate shredding, we compute the vector representation from the indexed form of our example of type `[⟨Int,[Int]⟩]`. Applying $shred_{\langle\rangle}$ to the outer list yields the vector V_o:

$$V_o = \begin{array}{l} [\ (\langle\rangle, \langle\langle\rangle, 1\rangle, \langle 100, \langle\rangle\rangle)\ , \\ \ \ (\langle\rangle, \langle\langle\rangle, 2\rangle, \langle 200, \langle\rangle\rangle)\ , \\ \ \ (\langle\rangle, \langle\langle\rangle, 3\rangle, \langle 300, \langle\rangle\rangle)\ ] \end{array}$$

By shredding the three elements of the outer list with $\mathit{shred}_{\langle\langle\rangle,1\rangle}$, $\mathit{shred}_{\langle\langle\rangle,2\rangle}$ and $\mathit{shred}_{\langle\langle\rangle,3\rangle}$, respectively, we obtain the following vectors, each with a single segment:

$$
\begin{aligned}
V_{i\cdot1} &= \begin{bmatrix} (\langle\langle\rangle,1\rangle,\langle\langle\rangle,1\rangle,1\rangle,10), \\ (\langle\langle\rangle,1\rangle,\langle\langle\rangle,1\rangle,2\rangle,20), \\ (\langle\langle\rangle,1\rangle,\langle\langle\rangle,1\rangle,3\rangle,30) \end{bmatrix} \\
V_{i\cdot2} &= [\,] \\
V_{i\cdot3} &= \begin{bmatrix} (\langle\langle\rangle,3\rangle,\langle\langle\rangle,3\rangle,1\rangle,70), \\ (\langle\langle\rangle,3\rangle,\langle\langle\rangle,3\rangle,2\rangle,80) \end{bmatrix}
\end{aligned}
$$

The corresponding packages are $\texttt{[Int]}^{V_{i\cdot1}}$, $\texttt{[Int]}^{V_{i\cdot2}}$ and $\texttt{[Int]}^{V_{i\cdot3}}$. Each partial vector contains one individual segment for the respective inner list value. These three packages merge into the package

$$\texttt{[[Int]}^{V_{i\cdot1}} +\!\!+\, V_{i\cdot2} +\!\!+\, V_{i\cdot3}\,\texttt{]}^{V_o}$$

with two vectors V_o and $V_{i\cdot1} +\!\!+ V_{i\cdot2} +\!\!+ V_{i\cdot3}$. Our introductory examples did not include the top-level index $\langle\rangle$ in the inner index of the outermost list. Here, index composition starts with $\langle\rangle$ to obtain a uniform indexing scheme.

STITCHING LISTS For the reconstruction of nested lists from shredded packages we adopt function $\mathit{stitch}(-)$ directly from Cheney *et al.* [CLW14a]. Function $\mathit{stitch}(\rho)$ converts package ρ to a list value. List elements are reconstructed from vector elements by traversing vector payload values with the inner index of the current vector element. If the function encounters a nested list type, the segment associated with the current index is retrieved from the corresponding vector in the type annotation.

$$
\begin{aligned}
\mathit{stitch}(\texttt{[}\tau\texttt{]}^{V}) &= [\ \mathit{stitch}_{x.k}(\tau, x.p) \mid x \leftarrow V\ ] \\
\mathit{stitch}_s(\pi, p) &= p \\
\mathit{stitch}_s(\langle \ell_i : \tau_i \rangle_{i=1}^{n}, p) &= \langle \ell_i = \mathit{stitch}_s(\tau_i, p.\ell_i) \rangle_{i=1}^{n} \\
\mathit{stitch}_s(\texttt{[}\tau\texttt{]}^{V}, \langle\rangle) &= [\ \mathit{stitch}_{x.k}(\tau, x.p) \mid x \leftarrow V, s = x.s\ ]
\end{aligned}
$$

We have defined shredding to preserve the order of list elements in the order of segment elements. Stitching exploits this property to restore list elements in the original order: segment lookup is defined as a list comprehension that keeps the order of segment elements.

4.3.2 *Vector Operator Language*

So far, we have defined a flat representation of nested lists with index-based segment vectors. To express $\mathcal{FL}$ queries on this flat representation, we introduce the language $\mathcal{SL}$ of vector operators that work on segment vectors. Language $\mathcal{SL}$ is the intermediate representation from which code for backends is generated. It is designed to map well to the typical operations offered by a (relational) query engine. Before we define the language formally, we introduce the essential concepts based on examples.

In this introduction, we focus on data-parallel operators $p^{\uparrow}$ and replication operators ($\otimes$, $\otimes^{\uparrow}$). List type constructors in data types dictate the vector structure of the flat representation. Hence, we structure our discussion of data-parallel operations by categorizing them along their type signatures. First, though, we look at the crucial shape operations.

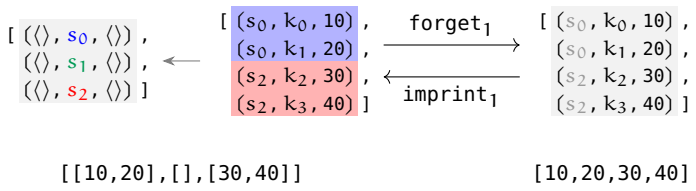


Figure 29: Shape operations on index-based vectors. Arrows → indicate a valid relationship between an inner and an outer vector.

SHAPE OPERATORS $\mathcal{FL}$ shape operations on index-based vectors work as outlined in Section 3.2.6. Lists are temporarily flattened by ignoring outer vectors. In Figure 29, we demonstrate $\texttt{forget}_1$ and $\texttt{imprint}_1$ on the index-based representation of list `[[10,20],[],[30,40]]`. Evaluating shape operations amounts to statically selecting the appropriate vectors during the generation of vector programs. No vector operators and thus no runtime effort is involved.

Note that the inner vector obtained via $\texttt{forget}_1$ preserves the segment structure of the inner vector instead of merging all elements into the unit segment. Thus, in contrast to the length-based representation, the inner vector is not the canonical encoding of the flat list `[10,20,30,40]`. Preserving the segment structure by preserving the inner vector unmodified is actually essential. Imagine an implementation of $\texttt{forget}_1$ that modifies the outer index to $\langle\rangle$. Then, we would have a shape operation that involves runtime effort. Worse, however, we would destroy the index relationship to the outer vector and thereby make a zero-cost implementation of $\texttt{imprint}_1$ impossible.

As it turns out, we can safely ignore the segment structure. Data-parallel $\mathcal{FL}$ operators $p^{\uparrow}$ uniformly apply to individual elements of a flat list of arguments. In the example of Figure 29, these are the elements of the inner vector. The segment structure of this vector does not play any role in the evaluation of data-parallel operators. Furthermore, we know that every occurrence of $\texttt{forget}_d$ in an $\mathcal{FL}$ expression generated by $\mathcal{L}[\![-]\!]$ is matched by a corresponding occurrence of $\texttt{imprint}_d$. Thus, outer vectors are always restored.

SCALAR OPERATORS Lifted scalar operators $c(\_,\ldots,\_)^{\uparrow}$ receive and produce lists of atomic values:

$$c(\_,\ldots,\_)^{\uparrow} : [\pi_1] \to \cdots \to [\pi_n] \to [\pi_r]$$

Each argument of type $[\pi]$ shreds to a package $[\pi]^V$ with a vector V that has an atomic payload of type π. Evaluating $c(\_,\ldots,\_)^{\uparrow}$ on the vector representation amounts to evaluating $c(\_,\ldots,\_)$ on the payload of corresponding vector elements. We have to (1) align corresponding vector elements (that is, elements with the same inner index), and (2) apply the scalar operation itself. In Figure 30, we see that these tasks are split among two separate $\mathcal{SL}$ operators: Operator `alignV` aligns two vectors based on their inner indexes and forms pairs of the respective payloads. Operator `projectV{_}` then applies an arbitrary scalar function to the payload of the resulting vector.

As vectors are aligned based on their inner indexes, they are required to have the same vector shape, *i.e.* the same indexes, segment structure and length. The translation into $\mathcal{SL}$ guarantees this property. Given two

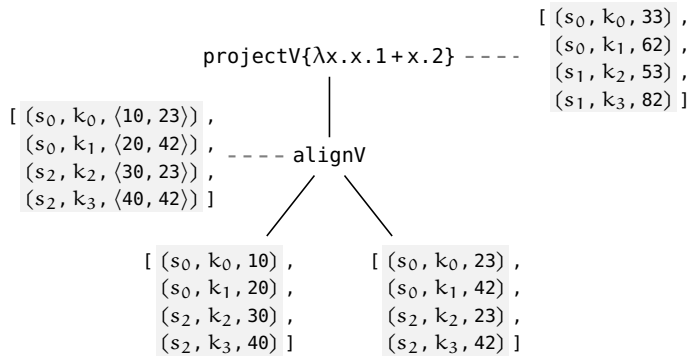


Figure 30: Implementing lifted addition $+^{\uparrow}$ in $\mathcal{SL}$. Dashed lines _ _ mark intermediate results of $\mathcal{SL}$ operators.

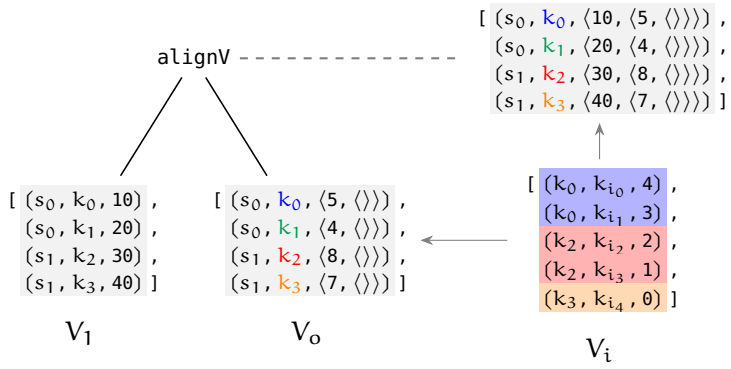


Figure 31: Implementing lifted pair construction $\langle \_, \_ \rangle^{\uparrow}$ on vectors.

vectors of equal shape, the result of `alignV` has the same shape. Likewise, `projectV{_}` does not change the shape of its operand.

For $c(\_, \ldots, \_)^{\uparrow}$, the individual operands are lists of atomic values, ensuring a representation with single vectors. Record operations, however, are typed more general:

$$\langle \ell_1 = \_, \ldots, \ell_n = \_ \rangle^{\uparrow} : [\tau_1] \to \cdots \to [\tau_n] \to [\langle \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \rangle]$$

Again, we obtain one vector for each argument that represents the argument list itself. However, $\tau_1, \ldots, \tau_n$ are arbitrary types that may include nested lists. In this case, the flat representation will include inner vectors. Does record construction affect inner vectors as well? Consider the lifted pair constructor $\langle e_1, e_2 \rangle^{\uparrow}$ with e_1 : `[Int]` and e_2 : `[⟨Int,[Int]⟩]`. Given the argument packages `[Int]`^{V_1} and `[⟨Int,[Int]`^{V_i}`⟩]`^{V_o}, the representation of the second operand consists of two vectors V_o and V_i. The situation is illustrated in Figure 31.

As pairs are constructed from individual elements of the argument lists, we use `alignV` to align the elements of the appropriate vectors. Again, both vectors are required to have equal shape and `alignV` maintains that shape. As the inner index of the result vector is identical to the inner index of V_o, the index relationship to the inner vector V_i remains valid. The inner vector is simply preserved in the result. Hence, record construction affects only the payload of the outer vectors.

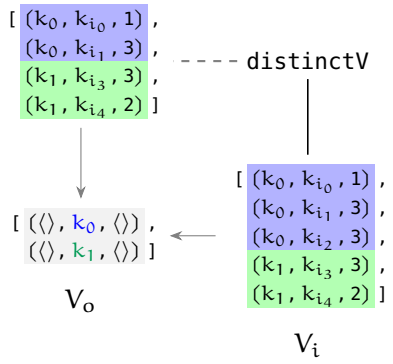


Figure 32: Implementing lifted duplicate elimination `distinct`$^{\uparrow}$ in $\mathcal{SL}$.

To summarize, scalar operations are executed on outer vectors. Their shape as well as any inner vectors are preserved.

SUSTAIN LIST NESTING DEPTH Next, we discuss list operations. Consider data-parallel duplicate elimination:

$$\texttt{distinct}^{\uparrow} : [[\delta]] \to [[\delta]]$$

Operators `sort`$^{\uparrow}$, `number`$^{\uparrow}$, `restrict`$^{\uparrow}$ and `append`$^{\uparrow}$ are similar in that they do not change the list nesting depth.

Duplicate elimination is performed on the individual inner lists, while the shape of the outer list is preserved. With the argument represented as package $[[\delta]^{V_i}]^{V_o}$, duplicate elimination is performed on the individual segments of the inner vector V_i, while the outer vector V_o is preserved (Figure 32).

Duplicate elimination is implemented by the vector operator `distinctV` that eliminates duplicates in the payload of a vector while observing segment boundaries. Since the outer index of the inner vector directly encodes segment boundaries, there is no need to consult the outer vector V_o. Note that `distinctV` preserves the outer index of V_i, so that the index relationship to V_o remains valid.

The implementation of duplicate elimination is the blueprint for all data-parallel list operators that sustain the list nesting depth of their argument: the operation is performed on the inner vector by vector operators on the segment level. So far, `append`$^{\uparrow}$ is the only list operator with two arguments (we will introduce more in Chapter 5). As in the case of scalar operators, we are guaranteed that the (outer) vectors of all arguments have equal shape, hence equal inner indexes. Accordingly, the inner vectors have compatible outer indexes. To implement `append`$^{\uparrow}$, vector operator `appendV` pairs corresponding segments of both inner vectors and appends the segments individually.

DECREASE LIST NESTING DEPTH Operators `reduce{}`$^{\uparrow}$ and `concat`$^{\uparrow}$ decrease the nesting depth of their argument:

$$\texttt{concat}^{\uparrow} : [[\tau]] \to [\tau]$$

Reducing list nesting amounts to the elimination of inner vectors. As an example, we show the $\mathcal{SL}$ implementation of `reduce{0, λx y.x+y}`$^{\uparrow}$ (*i.e.* `sum`$^{\uparrow}$) in Figure 33.

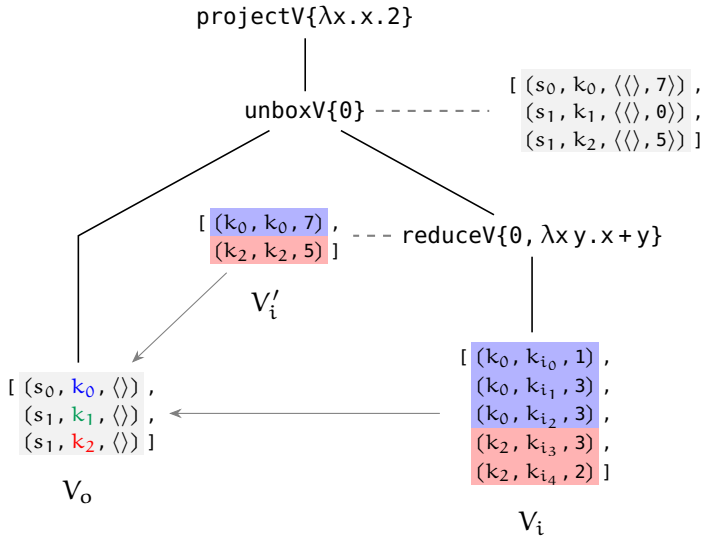


Figure 33: Implementing lifted aggregation in $\mathcal{SL}$.

In the example, summation is implemented by the $\mathcal{SL}$ operator `reduceV{}` that folds individual segments. The result V_i' of `reduceV{}` has the shape of the inner vector V_i. Together, V_o and V_i' describe a list of singleton lists ($[[\texttt{Int}]^{V_i'}]^{V_o}$). The result of $\texttt{sum}^{\uparrow}$, however, is a flat list of type `[Int]`, represented by a single vector. Operator `unboxV{}` joins outer and inner vectors along their indexes and places the payload of the inner vector's singleton segments in corresponding elements of the outer vector. Note that one segment is missing in the inner vectors V_i and V_i'. To account for missing segments, `unboxV{}` inserts a default payload value (here: `0` for the sum aggregate).

INCREASE LIST NESTING DEPTH Combinator $\texttt{group}^{\uparrow}$ increases list nesting:

$$\texttt{group}^{\uparrow} : [[\langle \tau, \delta \rangle]] \to [[\langle \delta, [\tau] \rangle]]$$

The singleton list constructor $\texttt{sng}^{\uparrow}$ belongs to the same category. Increasing list nesting amounts to the introduction of additional vectors. Hence, we need $\mathcal{SL}$ operators that introduce new vectors and establish a valid index relationship to them.

In Figure 34, we illustrate the shredding of $\texttt{group}^{\uparrow}$ into vector operators. Consider the expression $\texttt{group\{λx.x.2\}}^{\uparrow}\ e$ that groups argument e of type $[[\langle \texttt{Int}, \texttt{Text} \rangle]]$ based on the `Text` values. According to its type, e shreds to the package $[[\langle \texttt{Int}, \texttt{Text} \rangle]^{V_i}]^{V_o}$.

Again according to its type, the result of $\texttt{group\{\}}^{\uparrow}$ shreds into the following package:

$$[[\langle \texttt{Text}, [\langle \texttt{Int}, \texttt{Text} \rangle]^{V_g} \rangle]^{V_k}]^{V_o}$$

The outer vector V_o is identical to the outer vector of argument e. Figure 34 illustrates how the two inner vectors V_k and V_g are derived. Vector operator `groupV{}` constructs two vectors V_k and V_g from V_i. While V_k represents the *structure* of groups and contains the grouping keys, V_g represents the group *content* itself. Note that each segment of V_i is grouped individually. We have

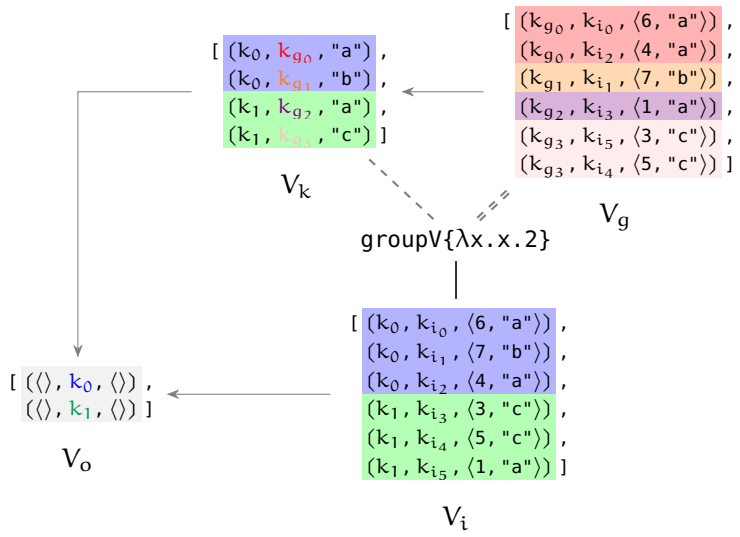


Figure 34: Implementing `group`$^{\uparrow}$ in $\mathcal{SL}$. Double and triple lines indicate the second and third vector result of an operator, respectively.

omitted simple projections on V_k and V_g that add $\langle\rangle$ as the placeholder for nested list values. While V_k preserves the outer indexes of V_i and is related to V_o, new indexes derived from the grouping key relate V_k and V_g. Note that vector V_g preserves the length and inner index of V_i. Only the order of elements and the segment structure changes.

REPLICATION We turn to the operators $\otimes$ and $\otimes^{\uparrow}$ used for data replication and environment lifting:

$$\otimes \quad : \quad [\tau_1] \to [\tau_2] \to [[\tau_1]]$$
$$\otimes^{\uparrow} \quad : \quad [\tau_1] \to [[\tau_2]] \to [[\tau_1]]$$

We illustrate the implementation of both operators in Figure 35.

As $\otimes$ increases list nesting, its $\mathcal{SL}$ counterpart `repV` creates a new vector V_{i_1} that replicates the elements of its left operand for each element of its right operand. `repV` establishes an index relationship between V_2 and V_{i_1} based on the inner index of V_2. Unique inner indexes for V_{i_1} are derived from the inner indexes of both V_1 and V_2. A simple projection derives the outer vector V_o from V_2.

Like `unboxV{}`, operator `repsegV` joins outer and inner vector based on their indexes. It propagates payload values from an outer vector (here: V_2) to an inner vector (here: V_{i_1}). With V_{i_2}, we have a vector that has the same shape as the inner vector produced by `repV`: outer and inner indexes concur, as well as vector length. The interplay of `repV` and `repsegV` reflects the construction of parallel environments in $\mathcal{FL}$ (Section 4.2): replication increases the nesting level of data, while environment lifting replicates data to this new nesting level.

PROPAGATING INDEX CHANGES The translation to $\mathcal{SL}$ maintains valid index relationships between vectors. In certain cases, we have to propagate changes to the shape of a vector to any inner vectors to keep the relationship valid. Consider the $\mathcal{FL}$ operator `restrict`, applied to an argument

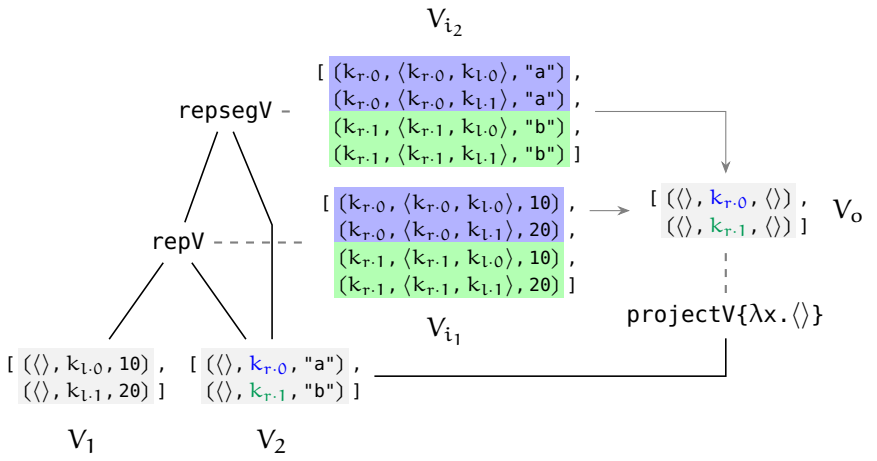


Figure 35: Implementing data replication ($\otimes$) and environment lifting ($\otimes^{\uparrow}$) in $\mathcal{SL}$.

with shredded package $[\langle \texttt{[Int]}^{V_i}, \texttt{Bool} \rangle]^{V_o}$. Figure $\mathcal{SL}$ illustrates its $\mathcal{SL}$ implementation.

The $\mathcal{SL}$ operator `selectV{}` filters V_o based on the second component of its payload. While the relationship of V_i to V_o is valid, it is not related to the outer result vector V_o': the outer index k_2 does not occur as an inner index in V_o'. The segment denoted by k_2 is no longer referenced. To re-establish a valid relationship, we eliminate the stale segment from V_i. Next to the vector V_o', `selectV{}` returns I, the description of a *filtering index transformation*. Transformation I is encoded as a simple list of valid index values. Operator `appfilterV` applies this transformation to V_i and removes any segments whose outer index does not occur in I.

Index propagation is recursive: `appfilterV` changes the shape of V_i. Accordingly, this change has to be propagated to any inner vectors that relate to V_i. Next to V_i', `appfilterV` returns a further index transformation that describes the change to the shape of V_i.

Other operators cause index transformations as well and not all of them are of the same form. We identify four different forms of index transformations:

1. If elements are removed from a vector, a filtering transformation lists the inner indexes of the remaining elements. For a vector with inner index type δ, the filtering transformation is of type $[\delta]$. In the example of Figure 36, the filtering transformation is $[k_0, k_1, k_3]$.

2. Operator `repsegV` replicates elements of a vector. Given the application of `repsegV` to V_2 and V_{i_1} in Figure 35, `repsegV` replicates elements of V_2 and maps them to the indexes of V_{i_1}. In any inner vector that is related to V_2, segments need to be replicated and mapped to the new indexes. The segment identified by $k_{r\cdot 0}$ needs to be replicated into two segments identified by $\langle k_{r\cdot 0}, k_{l_c \texttt{dot0}} \rangle$ and $\langle k_{r\cdot 0}, k_{l_c \texttt{dot1}} \rangle$.

   A *replication transformation* describes the changes to segments: it maps segment identifiers to new segment identifiers. In the example in Fig-

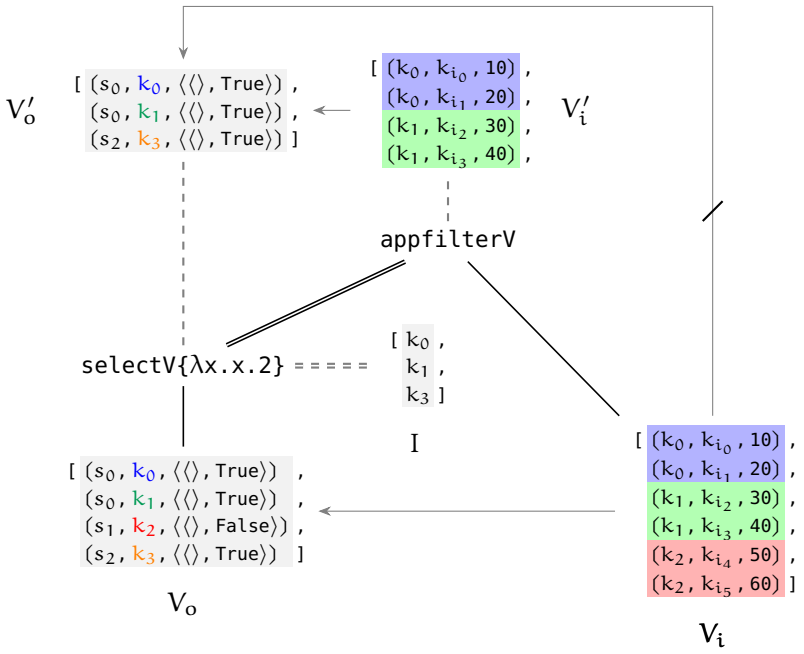


Figure 36: Implementing `restrict` in $\mathcal{SL}$. Propagating index changes to V_i maintains the index relationship to the outer vector.

ure 35, operator `repsegV` generates the following replication transformation:

$$[\ \langle f = k_{r\cdot 0}, t = \langle k_{r\cdot 0}, k_{l\cdot 0}\rangle\rangle\ ,\ \langle f = k_{r\cdot 0}, t = \langle k_{r\cdot 0}, k_{l\cdot 1}\rangle\rangle\ ,\ \langle f = k_{r\cdot 1}, t = \langle k_{r\cdot 1}, k_{l\cdot 0}\rangle\rangle\ ,\ \langle f = k_{r\cdot 1}, t = \langle k_{r\cdot 1}, k_{l\cdot 1}\rangle\rangle\ ]$$

Both segments identified by $k_{r\cdot 0}$ and $k_{r\cdot 1}$ are replicated into two copies identified by the new index of V_{i_2}.

3. If elements of a vector with inner index type t are reordered (*e.g.* in the implementation of `sort`$^{\uparrow}$ and `group`$^{\uparrow}$), a *sorting transformation* reorders segments in any inner vectors. For example, operator `groupV{}` in Figure 34 re-orders elements of the inner vector and generates the following sorting transformation:

$$[k_{i_0}, k_{i_2}, k_{i_1}, k_{i_5}, k_{i_3}, k_{i_4}]$$

4. As a special form of a replication transformation, a *rekeying transformation* describes a one-to-one mapping from old indexes to new indexes. Applying a rekeying transformation to an inner vector does not change the segment structure. Only the outer index is updated.

Distinguishing different forms of index transformation is not strictly necessary. Replication transformations as the most general type subsume the three other forms we have described. For example, a filtering transformation of type $[\delta]$ could be expressed as a replication transformation $[\langle f{:}\delta, t{:}\delta\rangle]$ that omits certain index values. Making the distinction, however, allows us to generate more specific and efficient backend code. Different forms of index transformations differ in their runtime cost depending on the concrete backend. Sorting transformations, for example, are necessary in a backend

Statements, Tuple Patterns

$bs ::= p \leftarrow o\ x \cdots x; bs \mid p \leftarrow o\ x \cdots x$

$p ::= x \mid (x,x) \mid (x, x, x)$

Vector Operators

```
o ::= tableV{t} | litV{[v...,v]} | projectV{f} | selectV{s}
    | numberV | sortV{s} | groupV{s} | distinctV | reduceV{s, s}
    | appendV | unsegmentV | mergesegV | alignV | unboxV{s}
    | repV | repseg V | combineV
    | apprepV | appfilterV | appsortV | appkeyV
```

Figure 37: Syntax of $\mathcal{SL}$ programs.

that keeps vectors physically ordered (Section 4.3.2.3). Applying sorting transformations in this setting involves actual work. Lowering vectors to unordered multisets, on the other hand, eliminates the necessity for sorting transformations altogether (Chapter 6).

4.3.2.1 *Formal Definition of $\mathcal{SL}$*

Having introduced the essential concepts of operations on flat vectors, we now introduce the vector language $\mathcal{SL}$ completely and formally. Figure 37 defines the syntax of vector programs. We adopt notation used by Blelloch [Ble90], and Suciu and Tannen [ST94], and define $\mathcal{SL}$ programs as non-empty sequences of statements. A statement $x \leftarrow o\ x \cdots x$ applies operator o to the variables x_1 to x_n bound by preceding statements. The result of o is bound to variable x. For example, the $\mathcal{SL}$ implementation of $\texttt{sum}^{\uparrow}$ (Figure 33) is expressed as the following vector program:

$$
\begin{aligned}
V &\leftarrow \texttt{reduceV\{0, } \lambda x\, y.\, x + y\texttt{\}}\ V_i \\
V' &\leftarrow \texttt{unboxV\{0\}}\ V_o\ V \\
V_r &\leftarrow \texttt{projectV\{}\lambda x.\, x.2\texttt{\}}
\end{aligned}
$$

As vector operators may return multiple results, a statement can match on a tuple and bind multiple names at once. Filtering inner and outer vectors (Figure 36) is expressed as the following vector program:

$$
\begin{aligned}
(V_r, V_t) &\leftarrow \texttt{selectV\{}\lambda x.\, x.1\texttt{\}}\ V_o \\
(V_i', V_t') &\leftarrow \texttt{appfilterV}\ V_t\ V_i
\end{aligned}
$$

In $\mathcal{FL}$ all computations are expressed as operators on the same level. In contrast, computations in $\mathcal{SL}$ are expressed on two different levels: vector operators express computations on lists, while scalar computations are exclusively expressed in the arguments of vector operators. Vector operators are parameterized with scalar functions s: `sortV{s}` applies s to the payload of each vector element to obtain a sorting key.

Vector operators belong to the following categories.

- Base operators `tableV{}` and `litV{}` provide constant vectors.
- Operators `projectV{}` and `selectV{}` are vector counterparts of the usual relational algebra operators. `projectV{}` applies a scalar expression to the payload of each vector element and `selectV{}` filters vector elements based on the result of a scalar expression.

- Segmented $\mathcal{SL}$ operators express operations on individual segments. Segmented operators include `sortV{}` (sorting segments), `groupV{}` (grouping segments), `distinctV` (eliminating duplicates in segments) and `numberV` (enumerating elements of segments). Operator `reduceV{}` folds segments individually into single scalar values.
- Operators `repV` and `repsegV` provide global and per-segment data replication.
- A number of administrative operators explicitly maintain indexes and vector relationships. In preceding examples, we have described operators `alignV` and `unboxV{}`. Operators `segmentV` and `unsegmentV` modify the outer index of a vector. Every element is placed in the unit segment (`unsegmentV`) or in its own singleton segment (`segmentV`). Operator `mergesegV` merges the elements of an inner vector into the segment structure of an outer vector.
- Operator `combineV` is a direct counterpart of the $\mathcal{FL}$ operator of the same name. It merges two vectors into one.
- Operators `apprepV`, `appkeyV`, `appfilterV` and `appsortV` apply the various forms of index transformations.

We define a typing relation for $\mathcal{SL}$ in Figure 38. Rules $\mathcal{SL}$-TY-STMTS and $\mathcal{SL}$-TY-STMT derive typing contexts for vector programs (we have omitted the obvious rule variations for statements that bind multiple variables). The remaining rules derive result types of individual operators. In typing rules, we distinguish segment vectors and different kinds of index transformations.

Segment Vector

$V ::= \mathsf{D}\ (\alpha, \beta, \gamma)$

Index Transformations

$I ::= \mathsf{R}\ \beta\ \beta \mid \mathsf{K}\ \beta\ \beta \mid \mathsf{F}\ \beta \mid \mathsf{S}\ \beta$

Recall that α, β and γ range over *scalar* types. The type $\mathsf{D}\ (\alpha, \beta, \gamma)$ denotes a segment vector with outer index type α, inner index type β and payload type γ. Types $\mathsf{R}\ \beta_o\ \beta_n$ and $\mathsf{K}\ \beta_o\ \beta_n$ denote replication and rekeying transformations mapping from index type β_o to β_n. Finally, $\mathsf{F}\ \beta$ and $\mathsf{S}\ \beta$ are the types of filtering and sorting transformations for indexes of type β.

In general, indexes are non-uniform and vectors may have indexes of different types. For operators with more than one operand, the typing rules specify the constraints that must be fulfilled by the index types. In an application of `alignV`, for example, the inner index type of both operands has to match (Rule $\mathcal{SL}$-TY-ALIGN). Operators like `repsegV` (Rule $\mathcal{SL}$-TY-REPSEG) and `unboxV{}` (Rule $\mathcal{SL}$-TY-UNBOX) that relate outer and inner vectors, on the other hand, require matching inner and outer indexes.

Most operators preserve indexes of their operands. Since `repV` V_1 V_2 replicates the elements of V_1, neither the inner index of V_1 nor V_2 identify elements of the result uniquely. Unique index values for the result are obtained by pairing index values from both inputs. Operators `apprepV` and `repsegV` also replicate elements and therefore incur the same effect. Grouping segments individually ($\mathcal{SL}$-TY-GROUP) produces a new vector in which the combination of segment identifier and grouping key uniquely identifies elements.

$$\frac{\Gamma \vdash x \leftarrow o\ x_1 \cdots x_n : V \qquad \Gamma, x : V \vdash bs \Rightarrow \Delta}{\Gamma \vdash x \leftarrow o\ x_1 \cdots x_n;\ bs \Rightarrow \{x : V\} \cup \Delta}\ \mathcal{SL}\text{-TY-STMTS}$$

$$\frac{\Gamma \vdash x \leftarrow o\ x_1 \cdots x_n : V}{\Gamma \vdash x \leftarrow o\ x_1 \cdots x_n; bs \Rightarrow \{x : V\}}\ \mathcal{SL}\text{-TY-STMT}$$

$$\frac{\Sigma(t) = \gamma \qquad \Phi(t) = \gamma_k}{\Gamma \vdash \texttt{tableV\{t\}} : D\ (\langle\rangle, \gamma_k, \gamma)}\ \mathcal{SL}\text{-TY-TABLE}$$

$$\frac{[\vdash s_i : \gamma]_{i=1}^{n}}{\Gamma \vdash \texttt{litV\{[s}_1\texttt{,...,s}_n\texttt{]\}} : D\ (\langle\rangle, \texttt{Int}, \gamma)}\ \mathcal{SL}\text{-TY-LIT}$$

$$\frac{\Gamma \vdash V : D\ (\alpha, \beta, \gamma) \qquad \vdash s : \gamma \to \gamma'}{\Gamma \vdash \texttt{projectV\{s\}}\ V : D\ (\alpha, \beta, \gamma')}\ \mathcal{SL}\text{-TY-PROJECT}$$

$$\frac{\Gamma \vdash V : D\ (\alpha, \beta, \gamma)}{\Gamma \vdash \texttt{distinctV}\ V : D\ (\alpha, \beta, \gamma)}\ \mathcal{SL}\text{-TY-DISTINCT}$$

$$\frac{\Gamma \vdash V : D\ (\alpha, \beta, \gamma) \qquad \vdash s : \gamma \to \texttt{Bool}}{\Gamma \vdash \texttt{selectV\{s\}}\ V : (D\ (\alpha, \beta, \gamma), F\ \beta)}\ \mathcal{SL}\text{-TY-SELECT}$$

$$\frac{\Gamma \vdash V : D\ (\alpha, \beta, \gamma) \qquad \vdash s : \gamma \to \gamma'}{\Gamma \vdash \texttt{sortV\{s\}}\ V : (D\ (\alpha, \beta, \gamma), S\ \beta)}\ \mathcal{SL}\text{-TY-SORT}$$

$$\frac{\Gamma \vdash V : D\ (\alpha, \beta, \gamma)}{\Gamma \vdash \texttt{numberV}\ V : D\ (\alpha, \beta, \langle\gamma, \texttt{Int}\rangle)}\ \mathcal{SL}\text{-TY-NUMBER}$$

$$\frac{\Gamma \vdash V : D\ (\alpha, \beta, \gamma) \qquad \vdash s_z : \delta \qquad \vdash s_f : \delta \to \gamma \to \delta}{\Gamma \vdash \texttt{reduceV\{s}_z\texttt{, s}_f\texttt{\}}\ V : D\ (\alpha, \alpha, \delta)}\ \mathcal{SL}\text{-TY-REDUCE}$$

$$\frac{\Gamma \vdash V_1 : D\ (\alpha, \beta_1, \gamma) \qquad \Gamma \vdash V_2 : D\ (\alpha, \beta_2, \gamma)}{\Gamma \vdash \texttt{appendV}\ V_1\ V_2 : D\ (\alpha, \texttt{Int}, \gamma)}\ \mathcal{SL}\text{-TY-APPEND}$$

$$\frac{\Gamma \vdash V_o : D\ (\alpha, \beta, \langle\rangle) \qquad \Gamma \vdash V_i : D\ (\beta, \beta', \gamma)}{\Gamma \vdash \texttt{mergesegV}\ V_o\ V_i : D\ (\alpha, \beta', \gamma)}\ \mathcal{SL}\text{-TY-MERGESEG}$$

$$\frac{\Gamma \vdash V_1 : D\ (\alpha, \beta, \gamma_1) \qquad \Gamma \vdash V_2 : D\ (\alpha, \beta, \gamma_2)}{\Gamma \vdash \texttt{alignV}\ V_1\ V_2 : D\ (\alpha, \beta, \langle\gamma_1, \gamma_2\rangle)}\ \mathcal{SL}\text{-TY-ALIGN}$$

$$\frac{\Gamma \vdash V : D\ (\alpha, \beta, \gamma)}{\Gamma \vdash \texttt{segmentV}\ V : D\ (\beta, \beta, \gamma)}\ \mathcal{SL}\text{-TY-SEGMENT}$$

$$\frac{\Gamma \vdash V : D\ (\alpha, \beta, \gamma)}{\Gamma \vdash \texttt{unsegmentV}\ V : D\ (\langle\rangle, \beta, \gamma)}\ \mathcal{SL}\text{-TY-UNSEGMENT}$$

$$\frac{\Gamma \vdash V_1 : D\ (\alpha, \beta_1, \gamma_1) \qquad \Gamma \vdash V_2 : D\ (\beta_1, \beta_2, \gamma_2) \qquad \vdash s : \gamma_2}{\Gamma \vdash \texttt{unboxV\{s\}}\ V_1\ V_2 : D\ (\alpha, \beta_1, \langle\gamma_1, \gamma_2\rangle)}\ \mathcal{SL}\text{-TY-UNBOX}$$

$$\frac{\Gamma \vdash V_1 : D\ (\langle\rangle, \beta_1, \gamma_1) \qquad \Gamma \vdash V_2 : D\ (\alpha, \beta_2, \gamma_2)}{\Gamma \vdash \texttt{repV}\ V_1\ V_2 : D\ (\beta_2, \langle\beta_2, \beta_1\rangle, \gamma_1)}\ \mathcal{SL}\text{-TY-REP}$$

$$\frac{\Gamma \vdash V_1 : D\ (\alpha_1, \beta_1, \gamma_1) \qquad \Gamma \vdash V_2 : D\ (\beta_1, \beta_2, \gamma_2)}{\Gamma \vdash \texttt{repsegV}\ V_1\ V_2 : (D\ (\beta_1, \beta_2, \gamma_1), R\ \beta_1\ \beta_2)}\ \mathcal{SL}\text{-TY-REPSEG}$$

$$\frac{\Gamma \vdash I : R\ \alpha_1\ \alpha_2 \qquad \Gamma \vdash V : D\ (\alpha_1, \beta, \gamma)}{\Gamma \vdash \texttt{apprepV}\ I\ V : (D\ (\alpha_2, \langle\alpha_2, \beta\rangle, \gamma), R\ \beta\ \langle\alpha_2, \beta\rangle)}\ \mathcal{SL}\text{-TY-APPREP}$$

$$\frac{\Gamma \vdash I : K\ \alpha_1\ \alpha_2 \qquad \Gamma \vdash V : D\ (\alpha_1, \beta, \gamma)}{\Gamma \vdash \texttt{appkeyV}\ I\ V : D\ (\alpha_2, \beta, \gamma)}\ \mathcal{SL}\text{-TY-APPKEY}$$

$$\frac{\Gamma \vdash I : S\ \alpha \qquad \Gamma \vdash V : D\ (\alpha, \beta, \gamma)}{\Gamma \vdash \texttt{appsortV}\ I\ V : (D\ (\alpha, \beta, \gamma), S\ \beta)}\ \mathcal{SL}\text{-TY-APPSORT}$$

$$\frac{\Gamma \vdash I : F\ \alpha \qquad \Gamma \vdash V : D\ (\alpha, \beta, \gamma)}{\Gamma \vdash \texttt{appfilterV}\ I\ V : (D\ (\alpha, \beta, \gamma), F\ \beta)}\ \mathcal{SL}\text{-TY-APPFILTER}$$

$$\frac{\Gamma \vdash V : D\ (\alpha, \beta, \gamma) \qquad \vdash s : \gamma \to \gamma'}{\Gamma \vdash \texttt{groupV\{s\}}\ V : (D\ (\alpha, \langle\alpha, \gamma'\rangle, \gamma'), D\ (\langle\alpha, \gamma'\rangle, \beta, \gamma), S\ \beta)}\ \mathcal{SL}\text{-TY-GROUP}$$

$$\frac{\Gamma \vdash V_b : D\ (\alpha, \beta, \texttt{Bool}) \qquad \Gamma \vdash V_t : D\ (\alpha, \beta, \gamma) \qquad \Gamma \vdash V_e : D\ (\alpha, \beta, \gamma)}{\Gamma \vdash \texttt{combineV}\ V_b\ V_t\ V_e : D\ (\alpha, \beta, \gamma)}\ \mathcal{SL}\text{-TY-COMBINE}$$

Figure 38: Typing rules for $\mathcal{SL}$ operators.

4.3.2.2 *Semantics of Vector Operators*

In order to define the semantics of vector programs and vector operators precisely, we define a list interpretation $\mathcal{S}[\![-]\!]_\rho$ of vector programs in terms of $\mathcal{ML}$.

We interpret sequences of statements by constructing an environment that maps variables to the result of vector operators (*i.e.* segment vectors and index transformations). For each statement, the (initially empty) environment ρ is extended with the respective binding. As for typing rules, we omit the trivial rule variations that handle statements binding multiple variables.

$$\mathcal{S}[\![x \leftarrow o\ x_1 \cdots x_n]\!]_\rho = \rho\,[x \mapsto \mathcal{S}[\![o\ x_1 \cdots x_n]\!]_\rho] \qquad (\mathcal{SL}\text{-STMT})$$

$$\mathcal{S}[\![x \leftarrow o\ x_1 \cdots x_n; bs]\!]_\rho = \qquad (\mathcal{SL}\text{-STMTS})$$
$$\mathcal{S}[\![bs]\!]_{\rho[x \mapsto V]} \quad \text{where } V = \mathcal{S}[\![o\ x_1 \cdots x_n]\!]_\rho$$

The interpretation of statements does not depend on the concrete list interpretation of vectors. We may plug in any alternative interpretation of vector operators. In Chapter 6, we define the lowering of vector programs to an unordered multiset algebra as such an alternative interpretation.

We now discuss briefly the various categories of $\mathcal{SL}$ operators. All operators are interpreted by mapping to list combinators and list comprehensions in $\mathcal{ML}$. As all $\mathcal{ML}$ constructs involved preserve the order of list elements, the order of vector elements and segments does not change. This enables us to translate list-based $\mathcal{FL}$ queries to $\mathcal{SL}$ programs without having to consider the preservation of list order explicitly. We first flatten data and care about the concrete implementation of list order afterwards when lowering to backend-specific collection types.

TABLE REFERENCES, LITERAL LISTS Tables references and literal lists are handled in a way identical to the indexed semantics of $\mathcal{FL}$ programs defined in Section 4.2.1. For base tables, both order and indexes are derived from their primary key. Indexes for literal tables are the enumeration of list elements.

$$\mathcal{S}[\![\texttt{tableV\{t\}}]\!]_\rho = \qquad (\mathcal{SL}\text{-TABLE})$$
$$[\ (\langle\rangle,\ \mathrm{pk}_t(x),\ x)\ |\ x \leftarrow \mathsf{sortWith}\ (\lambda x.\ \mathrm{pk}_t(x))\ [\![t]\!]\ ]$$
$$\mathcal{S}[\![\texttt{litV\{[}v_1, \ldots, v_n\texttt{]\}}]\!]_\rho = [(\langle\rangle, 1, [\![v_1]\!]), \ldots, (\langle\rangle, n, [\![v_n]\!])] \qquad (\mathcal{SL}\text{-LIT})$$

PROJECTION, SELECTION Simple list comprehensions suffice to implement `projectV{}` and `selectV{}`. Crucially, the order of vector elements does not change.

$$\mathcal{S}[\![\texttt{projectV\{s\}}\ V]\!]_\rho = [\ (x.s, x.k, [\![s]\!]\ x.p)\ |\ x \leftarrow \rho(V)\ ] \qquad (\mathcal{SL}\text{-PROJECT})$$

$$\mathcal{S}[\![\texttt{selectV\{s\}}\ V]\!]_\rho = \qquad (\mathcal{SL}\text{-SELECT})$$
$$\mathsf{let}\ vr = [\ x\ |\ x \leftarrow \rho(V), [\![s]\!]\ x.p\ ]$$
$$\quad m = [\ x.k\ |\ x \leftarrow vr\ ]$$
$$\mathsf{in}\ (vr, m)$$

SEGMENTED OPERATORS The interpretation of segmented operators follows a uniform pattern: we first compute an explicit list of segments from

the vector. From the list representation of a vector, combinator segs computes a list of pairs of outer indexes and corresponding segments:

$$\begin{aligned}\mathsf{segs}\ \mathsf{xs} = [\ \langle \mathsf{r}, [\ \mathsf{x} \mid \mathsf{x} \leftarrow \mathsf{xs}, \mathsf{x.s} = \mathsf{r}\ ]\rangle \\ \quad \mid \mathsf{r} \leftarrow \mathsf{nub}\ [\ \mathsf{x.s} \mid \mathsf{x} \leftarrow \mathsf{xs}\ ]\end{aligned}$$

Note that segs preserves the order of segments as well as the order of individual vector elements. The actual computation is then performed on each segment individually by mapping the corresponding $\mathcal{ML}$ combinator over the list of segments.

$$\mathcal{S}[\![\texttt{distinctV}\ V]\!]_\rho = \qquad (\mathcal{SL}\text{-DISTINCT})$$
$$\quad \mathsf{concat}\ [\ \mathsf{nubWith}\ \pi_\mathsf{p}\ \mathsf{seg.2} \mid \mathsf{seg} \leftarrow \mathsf{segs}\ \rho(V)\ ]$$
$$\mathcal{S}[\![\texttt{numberV}\ V\ ]\!]_\rho = \qquad (\mathcal{SL}\text{-NUMBER})$$
$$\begin{aligned}&\quad \mathsf{concat}\ [\ [\ (\mathsf{x.1.s}, \mathsf{x.1.k}, \langle \mathsf{x.1.p}, \mathsf{x.2}\rangle)\ \mid \mathsf{x} \leftarrow \mathsf{enum\ seg.2}\ ] \\ &\qquad\quad \mid \mathsf{seg} \leftarrow \mathsf{segs}\ \rho(V)\ ]\end{aligned}$$
$$\mathcal{S}[\![\texttt{sortV\{}s\texttt{\}}\ V]\!]_\rho = \qquad (\mathcal{SL}\text{-SORT})$$
$$\begin{aligned}&\quad \mathsf{let}\ \mathsf{vr} = \mathsf{concat}\ [\ \mathsf{sortWith}\ (\llbracket s \rrbracket \circ \pi_\mathsf{p})\ \mathsf{seg.1} \mid \mathsf{seg} \leftarrow \mathsf{segs}\ \rho(V)\ ] \\ &\qquad \mathsf{m} = [\ \mathsf{x.k} \mid \mathsf{x} \leftarrow \mathsf{vr}\ ] \\ &\quad \mathsf{in}\ (\mathsf{vr}, \mathsf{m})\end{aligned}$$
$$\mathcal{S}[\![\texttt{reduceV\{}s_z\texttt{,}\ s_f\texttt{\}}\ V]\!]_\rho = \qquad (\mathcal{SL}\text{-REDUCE})$$
$$\quad [\ (\mathsf{seg.1}, \mathsf{seg.1}, \mathsf{foldl}\ (\llbracket s_f \rrbracket\ \oslash\ \pi_\mathsf{p})\ \llbracket s_z \rrbracket\ \mathsf{seg.2}) \mid \mathsf{seg} \leftarrow \mathsf{segs}\ \rho(V)\ ]$$

Although it follows the same pattern as other segmented operators, the interpretation of `groupV{}` stands out in being somewhat complex. The complexity stems from the fact that by grouping individual segments, we obtain a list of groups g for each segment, *i.e.* increase the level of list nesting. Each segment of the original input is split into a number of segments. Rule $\mathcal{SL}$-GROUP explicitly derives the corresponding vector representation: the contents of all groups are consolidated in one inner vector (vi) and an outer vector (vo) is established with a valid index relationship based on original outer indexes and grouping keys.

$$\mathcal{S}[\![\texttt{groupV\{}s\texttt{\}}\ V]\!]_\rho = \qquad (\mathcal{SL}\text{-GROUP})$$
$$\begin{aligned}&\quad \mathsf{let}\ \mathsf{gs} = [\ \langle\langle \mathsf{seg.1}, \mathsf{g.1}\rangle, \mathsf{g.2}\rangle \\ &\qquad\quad \mid \mathsf{seg} \leftarrow \mathsf{segs}\ \rho(V), \mathsf{g} \leftarrow \mathsf{groupWith}\ (\llbracket s \rrbracket \circ \pi_\mathsf{p})\ \mathsf{seg.2}\ ] \\ &\qquad \mathsf{vo} = [\ (\mathsf{x.1.1}, \mathsf{x.1}, \mathsf{x.1.2})\ \mid \mathsf{x} \leftarrow \mathsf{gs}\ ] \\ &\qquad \mathsf{vi} = \mathsf{concat}\ [\ [\ (\mathsf{g.1}, \mathsf{x.k}, \mathsf{x.p})\ \mid \mathsf{x} \leftarrow \mathsf{g.1.3}\ ] \\ &\qquad\qquad\quad \mid \mathsf{g} \leftarrow \mathsf{gs}\ ] \\ &\qquad \mathsf{m} = [\ \mathsf{x.k} \mid \mathsf{x} \leftarrow \mathsf{vi}\ ] \\ &\quad \mathsf{in}\ (\mathsf{vo}, \mathsf{vi}, \mathsf{m})\end{aligned}$$

$$\mathcal{S}[\![\texttt{appendV}\ V_1\ V_2]\!]_\rho = \qquad (\mathcal{SL}\text{-APPEND})$$
$$\begin{aligned}&\quad \mathsf{let}\ \mathsf{i1} = [\ \langle (\mathsf{x.1.s}, \langle 1, \mathsf{x.2}\rangle, \mathsf{x.1.p}), \mathsf{x.1.k}\rangle \mid \mathsf{x} \leftarrow \mathsf{enum}\ \rho(V_1)\ ] \\ &\qquad \mathsf{m1} = [\ \langle \mathsf{f} = \mathsf{x.2}, \mathsf{t} = \mathsf{x.1.k}\rangle \mid \mathsf{x} \leftarrow \mathsf{i1}\ ] \\ &\qquad \mathsf{i2} = [\ \langle (\mathsf{x.1.s}, \langle 2, \mathsf{x.2}\rangle, \mathsf{x.1.p}), \mathsf{x.1.k}\rangle \mid \mathsf{x} \leftarrow \mathsf{enum}\ \rho(V_2)\ ] \\ &\qquad \mathsf{m2} = [\ \langle \mathsf{f} = \mathsf{x.2}, \mathsf{t} = \mathsf{x.1.k}\rangle \mid \mathsf{x} \leftarrow \mathsf{i2}\ ] \\ &\qquad \mathsf{va} = \mathsf{sortWith}\ (\lambda \mathsf{x}.\langle \mathsf{x.s}, \mathsf{x.p}\rangle)\ (\mathsf{i1}\ +\!\!+\ \mathsf{i2}) \\ &\quad \mathsf{in}\ (\mathsf{va}, \mathsf{m1}, \mathsf{m2})\end{aligned}$$

REPLICATION Replication of vectors (Rule $\mathcal{SL}$-REP) and replication of individual elements per segment (Rule $\mathcal{SL}$-REPSEG) is implemented through list comprehensions with multiple generators. While Rule $\mathcal{SL}$-REP copies a complete vector, Rule $\mathcal{SL}$-REPSEG restricts to matching pairs of inner and outer index. Note that the order of generators in the main comprehension of Rule $\mathcal{SL}$-REP keeps the individual copies of the segment from V_1 dense: elements of different segments are not mixed.

$$\mathcal{S}[\![\texttt{repV}\ V_1\ V_2]\!]\rho = [\ (x.k, \langle x.k, y.k\rangle, y.p) \mid x \leftarrow \rho(V_2), y \leftarrow \rho(V_1)\ ] \qquad (\mathcal{SL}\text{-REP})$$

$$\mathcal{S}[\![\texttt{repsegV}\ V_1\ V_2]\!]\rho = \qquad (\mathcal{SL}\text{-REPSEG})$$
$$\begin{aligned}
&\texttt{let}\ vp = [\ \langle (x.s, x.k, y.p), y.k\rangle \\
&\qquad\qquad \mid x \leftarrow \rho(V_2), y \leftarrow \rho(V_1), x.k = y.s\ ] \\
&\quad v = [\ x.1 \mid x \leftarrow vp\ ] \\
&\quad m = [\ \langle f = x.1, t = x.1.k\rangle \mid x \leftarrow vp\ ] \\
&\texttt{in}\ (v, m)
\end{aligned}$$

ADMINISTRATIVE OPERATORS Except for `unboxV{}`, all administrative operators are defined through simple list comprehensions. Based on the index of the outer vector, Rule $\mathcal{SL}$-UNBOX determines any empty or missing segments in the inner vector and inserts the default value for them. Note that Rule $\mathcal{SL}$-UNBOX critically relies on singleton segments in the inner vector.

$$\mathcal{S}[\![\texttt{mergesegV}\ V_1\ V_2]\!]\rho = \qquad (\mathcal{SL}\text{-MERGESEG})$$
$$[\ (x.s, y.k, y.p) \mid x \leftarrow \rho(V_1), y \leftarrow \rho(V_2), x.k = y.s\ ]$$

$$\mathcal{S}[\![\texttt{alignV}\ V_1\ V_2]\!]\rho = \qquad (\mathcal{SL}\text{-ALIGN})$$
$$[\ (x.s, x.k, \langle x.p, y.p\rangle) \mid x \leftarrow \rho(V_1), y \leftarrow \rho(V_2), x.k = y.k\ ]$$

$$\mathcal{S}[\![\texttt{segmentV}\ V]\!]\rho = [\ (x.k, x.k, x.p) \mid x \leftarrow \rho(V)\ ] \qquad (\mathcal{SL}\text{-SEGMENT})$$

$$\mathcal{S}[\![\texttt{unsegmentV}\ V]\!]\rho = [\ (\langle\rangle, x.k, x.p) \mid x \leftarrow \rho(V)\ ] \qquad (\mathcal{SL}\text{-UNSEGMENT})$$

$$\mathcal{S}[\![\texttt{unboxV\{s\}}\ V_o\ V_i]\!]\rho = \qquad (\mathcal{SL}\text{-UNBOX})$$
$$\begin{aligned}
&\texttt{concat}\ [\ \texttt{if null r} \\
&\qquad\qquad \texttt{then}\ [(x.s, x.k, \langle x.p, [\![s]\!]\rangle)] \\
&\qquad\qquad \texttt{else}\ [\ (x.s, x.k, \langle x.p, y.p\rangle) \mid y \leftarrow \texttt{seg.2}\ ] \\
&\qquad\quad \mid x \leftarrow \rho(V_o), \texttt{seg} \leftarrow \texttt{segs}\ \rho(V_i), x.k = \texttt{seg.2}\ ]
\end{aligned}$$

$$\mathcal{S}[\![\texttt{combineV}\ V_b\ V_1\ V_2]\!]\rho = \qquad (\mathcal{SL}\text{-COMBINE})$$
$$\begin{aligned}
&[\ (x.s, x.k, y.p) \\
&\mid x \leftarrow \rho(V_b), y \leftarrow \rho(V_1)\ +\!\!+\ \rho(V_2), x.k = y.k\ ]
\end{aligned}$$

INDEX TRANSFORMATIONS

$$\mathcal{S}[\![\texttt{apprepV I V}]\!]_\rho = \qquad (\mathcal{SL}\text{-APPREP})$$
$$\quad \text{let } \mathsf{vp} = [\ \langle\langle \mathsf{x.t}, \mathsf{y.k}\rangle, \mathsf{y.p}\rangle \mid \mathsf{x} \leftarrow \rho(\mathrm{I}), \mathsf{y} \leftarrow \rho(\mathrm{V}), \mathsf{y.f} = \mathsf{x.s}\ ]$$
$$\quad\quad \mathsf{v} = [\ (\mathsf{x.1.1}, \mathsf{x.1}, \mathsf{x.2}) \mid \mathsf{x} \leftarrow \mathsf{vp}\ ]$$
$$\quad\quad \mathsf{m} = [\ \langle \mathsf{f} = \mathsf{x.1.2}, \mathsf{t} = \mathsf{x.1}\rangle \mid \mathsf{x} \leftarrow \mathsf{vp}\ ]$$
$$\quad \text{in } (\mathsf{v}, \mathsf{m})$$
$$\mathcal{S}[\![\texttt{appkeyV I V}]\!]_\rho = \qquad (\mathcal{SL}\text{-APPKEY})$$
$$\quad [\ (\mathsf{x.t}, \mathsf{y.k}, \mathsf{y.p}) \mid \mathsf{x} \leftarrow \rho(\mathrm{I}), \mathsf{y} \leftarrow \rho(\mathrm{V}), \mathsf{x.f} = \mathsf{y.s}\ ]$$
$$\mathcal{S}[\![\texttt{appfilterV I V}]\!]_\rho = \qquad (\mathcal{SL}\text{-APPFILTER})$$
$$\quad \text{let } \mathsf{vp} = [\ \mathsf{y} \mid \mathsf{x} \leftarrow \rho(\mathrm{I}), \mathsf{y} \leftarrow \rho(\mathrm{V}), \mathsf{x} = \mathsf{y.s}\ ]$$
$$\quad\quad \mathsf{m} = [\ \mathsf{x.k} \mid \mathsf{x} \leftarrow \mathsf{vp}\ ]$$
$$\quad \text{in } (\mathsf{v}, \mathsf{m})$$
$$\mathcal{S}[\![\texttt{appsortV I V}]\!]_\rho = \mathcal{S}[\![\texttt{appfilterV I V}]\!]_\rho \qquad (\mathcal{SL}\text{-APPSORT})$$

4.3.2.3 *Implementing Vector Operators*

The list interpretation $\mathcal{S}[\![-]\!]$ of $\mathcal{SL}$ operators not only defines their semantics. It also outlines the constructs required of concrete backends to evaluate $\mathcal{SL}$, and thus the nested calculus $\mathcal{CL}$. Despite the rather large and specific set of operators comprising $\mathcal{SL}$, these requirements are modest. All $\mathcal{SL}$ operators can be expressed with comprehensions over flat collections, complemented with a couple of list combinators (*e.g.* sortWith).

All inputs and results produced by $\mathcal{S}[\![-]\!]$ are flat. From the perspective of query languages, the comprehension-based implementation of $\mathcal{SL}$ operators matches familiar calculus patterns that express joins and projections. Hence, we can expect that most $\mathcal{SL}$ operators map directly to relational operators.

Only segmented list operators do not fit this picture. In the implementation of those, we use nested intermediate results to represent segments explicitly. This is not mandatory and does not hinder lowering to a backend with flat collections. In defining $\mathcal{S}[\![-]\!]$, we have restricted ourselves to list combinators of $\mathcal{ML}$ to be on common semantical ground with the definition of $\mathcal{CL}$ and $\mathcal{FL}$. Explicitly materializing segments allows the implementation of segmented operations with those regular list operations. Actual backend implementations can use efficient implementations of individual segment operators that do not materialize nested data. In Chapter 6, for example, we demonstrate that the bulk processing primitives of flat relational algebra are perfectly sufficient to express all segmented operations efficiently.

We consider multiple implementation alternatives for $\mathcal{SL}$. Among other things, the representation of list order characterizes potential backends.

1. We may lower flat, ordered vectors to unordered multisets and translate $\mathcal{SL}$ to relational algebra. Following this way, we can generate SQL code and target relational query engines. A key challenge in this setting is to preserve the order semantics of $\mathcal{SL}$ on unordered multisets while minimizing the order constraints in relational queries. In the remainder of this thesis, we follow this route (Chapter 6).

2. We may closely adhere to the order semantics of $\mathcal{SL}$ and lower vectors to physically ordered collections. Order-preserving backend primitives can preserve the order of vector elements without additional effort. Segmented $\mathcal{SL}$ operations can be implemented with existing

primitives. Alternatively, we may add specialized primitives that directly support individual segmented operations efficiently and take the specifics of the vector model into account.

Advised by the author of this thesis, Bruder [Bru17] describes an implementation of $\mathcal{SL}$ on top of MonetDB's MAL [MKB09] algebra. The code generator maps ordered segment vectors to MonetDB's ordered BATs. It exploits the order-preserving nature of MAL operators to maintain that physical order without sorting effort. Certain $\mathcal{SL}$ operators (*e.g.* `alignV`) then come for free. Furthermore, the dense storage of segments can be exploited for some operations: if elements of different segments are not interleaved, a single scan over a BAT is sufficient to implement `reduceV{}`, for example. Bruder compares the runtime of MAL code generated with this backend to SQL:2003 code generated with the relational backend described in Chapter 6. For some queries, the MAL backend is considerably faster.

There is no reason to target only full-blown relational database management systems. Recently, a number of low-level intermediate query representations have been proposed that serve as a starting point for compilation to native code: ScaLite [Sha+16], Voodoo [Pir+16] and Weld [Pal+17]. These intermediate representations are centered around flat lists and arrays and could be targetted to enable compiled in-memory execution of $\mathcal{SL}$ programs.

4.3.3 *Shredding $\mathcal{FL}$ Expressions*

So far, we have defined segment vectors and the shredding of list values into vectors. We have also defined $\mathcal{SL}$, a language over segment vectors. Now, we add the last building block of *Query Flattening*: the shredding of $\mathcal{FL}$ expressions. For each construct of $\mathcal{FL}$, we define its implementation in terms of a sequence of $\mathcal{SL}$ operators on the flat representation.

TRANSLATION BY TYPING To organize the translation, we exploit the direct correspondence between a list type and the vector structure of its flat representation. Translation to $\mathcal{SL}$ takes the form of inference rules that define a relation $e \Rightarrow \rho$ to derive a shredded package ρ for expression e. The package ρ has the same structure as the regular type inferred according to rules in Figure 27. List type constructors in shredded packages are annotated with vector variable names. As a side effect, shredding rules emit sequences of $\mathcal{SL}$ statements that bind the vector variables.

As an example, consider the shredding rules for literal lists (SHRED-LIT) and table references (SHRED-TABLE). Each rule emits one $\mathcal{SL}$ statement. We enclose the sequence of statements emitted in $[\cdot]$ brackets. The flat list type inferred for both constructs is annotated with the name bound by the respective $\mathcal{SL}$ operator.

$$\textsc{shred-table}\ \frac{[V \leftarrow \texttt{tableV\{t\}}]}{\Gamma \vdash \texttt{table(t)} \Rightarrow [\Sigma(t)]^V} \qquad \textsc{shred-lit}\ \frac{[\,\vdash V_i : \pi]_{i=1}^n \qquad [V \leftarrow \texttt{litV\{[}v_1\texttt{,}\ldots\texttt{,}v_n\texttt{]\}}]}{\Gamma \vdash \texttt{[}v_1\texttt{,}\ldots\texttt{,}v_n\texttt{]} \Rightarrow [\pi]^V}$$

Statements emitted by shredding rules are appended to a global sequence of statements that is initially empty. As type inference rules describe a bottom-up traversal of expressions, we obtain a linear sequence of $\mathcal{SL}$ statements that describe the $\mathcal{SL}$ program and compute all vectors that make up

the query result. We assume all variable names bound by vector statements to be unique, such that no shadowing occurs.

EVALUATING SHREDDED TERMS The result of lowering a $\mathcal{FL}$ expression consists of two components: (1) the complete vector program emitted by shredding rules, together with (2) the inferred shredded package that references individual results of that vector program. Together, these provide all information required to evaluate a shredded $\mathcal{FL}$ expression and assemble the nested list result from flat vectors. Evaluating a vector program with $\mathcal{S}[\![-]\!]_\rho$ (Section 4.3.2.2) results in an environment ρ that maps each name bound in the vector program to a vector or index transformation value, *i.e.* a flat list. Traversing the shredded type and replacing each vector name with the corresponding vector value results in a shredded package from which the nested result can be reconstructed with $stitch(-)$. Any alternative interpretation of vector programs can be plugged in instead of $\mathcal{S}[\![-]\!]_\rho$ to execute flattened queries on actual backends (*e.g.* Chapter 6).

PLAN SHARING In introductory $\mathcal{SL}$ examples, we have seen the result of individual operators being referenced multiple times (*e.g.* Figures 34 to 36). In general, shredding of $\mathcal{FL}$ expressions exposes sharing and the resulting vector programs are DAG-shaped. Information about sharing can be exploited during backend code generation to reuse intermediate results. Some instances of sharing are observable in the $\mathcal{FL}$ term (variables being referenced multiple times), others are exposed by shredding rules (*e.g.* operators like `groupV{}` with multiple results). In other cases, independent rule applications emit the same operators on the same arguments. In shredding rules, we consider sharing only to the extent described above. However, standard implementation techniques (*e.g.* hash consing) can be used to recover sharing of operator results completely and exploit it for backend code generation.

4.3.3.1 *Shredding Rules*

We present in turn shredding rules for the various categories of $\mathcal{FL}$ constructs. Together, these rules constitute a compositional translation from $\mathcal{FL}$ to $\mathcal{SL}$. $\mathcal{FL}$ operators are translated uniformly and independent of their enclosing expression.

BINDINGS, VARIABLES `let`-bindings are handled as in regular typing rules. The type for e_2 is inferred in a type context Γ extended with the shredded type inferred for e_1. Variable references are simple lookups in the type context.

$$\textsc{shred-let}\ \frac{\Gamma \vdash e_1 \Rightarrow \rho_1 \qquad \Gamma, x : \rho_1 \vdash e_2 \Rightarrow \rho_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow \rho_2} \qquad \textsc{shred-var}\ \frac{x : \rho \in \Gamma}{\Gamma \vdash x \Rightarrow \rho}$$

SHAPE OPERATIONS Shape operations $\texttt{forget}_d$ and $\texttt{imprint}_d$ work exclusively on the shredded package and do not emit any vector operators. Flattening the list structure comes down to removing the outer `d` list type constructors and returning the inner shredded list type constructor $\texttt{[t]}^V$. Restoring the list structure works analogously by re-adding the outer `d` annotated list type constructors.

$$\textsc{shred-forget}\quad \frac{\Gamma \vdash e \Rightarrow [\ldots[[\rho]^{V}]^{V_d}\ldots]^{V_l}}{\Gamma \vdash \texttt{forget}_d\ e \Rightarrow [\rho]^{V}}$$

$$\textsc{shred-imprint}\quad \frac{\Gamma \vdash e_1 \Rightarrow [\ldots[\rho_1]^{V_d}\ldots]^{V_l} \qquad \Gamma \vdash e_2 \Rightarrow [\rho_2]^{V}}{\Gamma \vdash \texttt{imprint}_d\ e_1\ e_2 \Rightarrow [\ldots[[\rho_2]^{V}]^{V_d}\ldots]^{V_l}}$$

SCALAR OPERATORS $\mathcal{SL}$ provides a binary `alignV` operator that aligns two vectors and constructs pairs of payload values. In the following rules, it is convenient to assume an n-ary `alignV` operator that aligns n vectors and constructs n-tuples of payload values. The latter operator can be easily implemented with $n-1$ applications of regular `alignV` and one `projectV{}`.

With Rule SHRED-BASE-OP, we shred data-parallel atomic primitives. We align the vectors $v_1, \ldots, v_n$ for all arguments based on their inner index and implement the actual operation with `projectV{}`.

$$\textsc{shred-base-op}\quad \frac{\Sigma(c(\_,\ldots,\_)) = \pi_1 \to \cdots \to \pi_n \to \pi \qquad [\Gamma \vdash e_i \Rightarrow [\pi_i]^{V_i}]_{i=1}^{n} \qquad \begin{bmatrix} V_t \leftarrow \texttt{alignV}\ V_1\ \cdots\ V_n \\ V \leftarrow \texttt{projectV\{}\lambda x.c(x.1,\ldots,x.n)\texttt{\}}\ V_t \end{bmatrix}}{\Gamma \vdash c(e_1,\ldots,e_n)^{\uparrow} \Rightarrow [\pi]^{V}}$$

Implementing record construction (Rule SHRED-RECORD) and record selection (Rule SHRED-RECORD-SEL) also comes down to aligning vectors and manipulating their payload. In both rules, we have to preserve any inner vectors. As inner vectors are organized in the packages of operands e_i, this happens automatically through the construction of the result package.

$$\textsc{shred-record}\quad \frac{[\Gamma \vdash e_i \Rightarrow [\rho_i]^{V_i}]_{i=1}^{n} \qquad \begin{bmatrix} V_t \leftarrow \texttt{alignV}\ V_1\ \cdots\ V_n \\ V \leftarrow \texttt{projectV\{}\lambda x.\langle \ell_i = x.i \rangle_{i=1}^{n}\texttt{\}}\ V_t \end{bmatrix}}{\Gamma \vdash \langle \ell_i = e_i \rangle_{i=1,\ldots,n}^{\uparrow} \Rightarrow [\langle \ell_i : \rho_i \rangle_{i=1}^{n}]^{V}}$$

$$\textsc{shred-record-sel}\quad \frac{\Gamma \vdash e \Rightarrow [\langle \ldots, \ell : t, \ldots \rangle]^{V_r} \qquad [V \leftarrow \texttt{projectV\{}\lambda x.x.\ell\texttt{\}}\ V_r]}{\Gamma \vdash e.\ell^{\uparrow} \Rightarrow [\rho]^{V}}$$

REPLICATION To implement the data replication operator $\otimes$, we employ its $\mathcal{SL}$ counterpart `repV` to construct the inner vector with replicated segments (Figure 35). The vector representation of operand e_2 serves as the template for the outer vector V_o. Only its payload has to be replaced with the placeholder value $\langle\rangle$ for inner lists.

$$\textsc{shred-dist}\quad \frac{\Gamma \vdash e_1 \Rightarrow [\delta]^{V_1} \qquad \Gamma \vdash e_2 \Rightarrow [\rho]^{V_2} \qquad \begin{bmatrix} V_o \leftarrow \texttt{projectV\{}\lambda x.\langle\rangle\texttt{\}}\ V_2 \\ V_i \leftarrow \texttt{repV}\ V_1\ V_2 \end{bmatrix}}{\Gamma \vdash e_1 \otimes e_2 \Rightarrow [[\delta]^{V_i}]^{V_o}}$$

The $\mathcal{SL}$ implementation of $\otimes^{\uparrow}$ with `repsegV` also has been demonstrated in Figure 35.

SHRED-DIST-LIFT

$$\frac{\Gamma \vdash e_1 \wr [\rho_1]^V \qquad \Gamma \vdash e_2 \wr [[\rho_2]^{V_i}]^{V_o} \qquad \left[(V_j, I) \leftarrow \texttt{repsegV}\ V\ V_i\right]}{\Gamma \vdash e_1 \otimes^{\uparrow} e_2 \wr [[\, (\!|\rho_1|\!)_I\,]^{V_j}]^{V_o}}$$

The index transformation I returned by `repsegV` describes the replication of elements of V. This transformation has to be applied to any inner vectors related to V. Inner vectors are found as annotations on list type constructors in the element package ρ_1. Function $(\!|-|\!)\ \_$ propagates a replicating index transformation recursively through a shredded package:

$$\begin{aligned}
&(\!|\pi|\!)_I = \pi \\
&(\!|\langle \ell_i : \rho_i \rangle_{i=1}^n|\!)_I = \langle \ell_i : (\!|\rho_i|\!)_I \rangle_{i=1}^n \\
&(\!|[\rho]^V|\!)_I = [\, (\!|\rho|\!)_{I'}\,]^{V'} \\
&\quad \left[(V', I') \leftarrow \texttt{apprepV}\ I\ V\right]
\end{aligned}$$

The special case of replicating a constant scalar value comes down to a simple projection.

SHRED-DIST-SCALAR

$$\frac{\vdash v : \pi \qquad \Gamma \vdash e \wr [\tau]^V \qquad \left[V' \leftarrow \texttt{projectV}\{\lambda x.v\}\ V\right]}{\Gamma \vdash \texttt{rep}\{v\}\ e \wr [\pi]^{V'}}$$

DATA-PARALLEL LIST OPS The implementation of $\texttt{restrict}^{\uparrow}$ has been sketched in Figure 36 and is defined in Rule SHRED-RESTRICT-LIFT. Operator `selectV{}` generates a filtering index transformation I that eliminates stale segments from inner vectors. We use function $(\!|-|\!)\ \_$ to propagate this transformation into ρ. Function $(\!|-|\!)\ \_$ emits operator `appfilterV` instead of `apprepV`, but is otherwise equivalent to $(\!|\_|\!)\ \_$.

SHRED-RESTRICT-LIFT

$$\frac{\Gamma \vdash e \wr [[\langle \rho, \texttt{Bool} \rangle]^{V_i}]^{V_o} \qquad \begin{bmatrix} (V_s, I) & \leftarrow \texttt{selectV}\{\lambda x.x.2\}\ V_i \\ V_p & \leftarrow \texttt{projectV}\{\lambda x.x.1\}\ V_s \end{bmatrix}}{\Gamma \vdash \texttt{restrict}^{\uparrow}\ e \wr [[\, (\!|\rho|\!)_I\,]^{V_p}]^{V_o}}$$

Most lifted list combinators are implemented as demonstrated in Figures 32 and 34: they map directly to segmented vector operators. Note that `groupV{}` (Rule SHRED-GROUP-LIFT) and `sortV{}` (Rule SHRED-SORT-LIFT) return sorting transformations. Function $\langle\!|-|\!\rangle\ \_$ propagates these transformations and is defined analogously to $(\!|-|\!)\ \_$. Sorting transformations are applied by operator `appsortV`.

$$\textsc{shred-distinct-lift}\quad \frac{\Gamma \vdash e \Rightarrow [[\delta]^{V_i}]^{V_o} \qquad [V \leftarrow \texttt{distinctV}\ V_i]}{\Gamma \vdash \texttt{distinct}^{\uparrow}\ e \Rightarrow [[\delta]^{V}]^{V_o}}$$

$$\textsc{shred-number-lift}\quad \frac{\Gamma \vdash e \Rightarrow [[\rho]^{V_i}]^{V_o} \qquad [V \leftarrow \texttt{numberV}\ V_i]}{\Gamma \vdash \#^{\uparrow}\ e \Rightarrow [[\langle \rho, \texttt{Int} \rangle]^{V}]^{V_o}}$$

$$\textsc{shred-sort-lift}\quad \frac{\Gamma \vdash e \Rightarrow [[\langle \rho, \delta \rangle]^{V_i}]^{V_o} \qquad \begin{bmatrix} (V_s, I) \leftarrow \texttt{sortV}\{\lambda x.x.2\}\ V_i \\ V \qquad\quad \leftarrow \texttt{projectV}\{\lambda x.x.1\}\ V_s \end{bmatrix}}{\Gamma \vdash \texttt{sort}^{\uparrow}\ e \Rightarrow [[\langle \rho \rangle_I]^{V}]^{V_o}}$$

$$\textsc{shred-group-lift}\quad \frac{\Gamma \vdash e \Rightarrow [[\langle \rho, \delta \rangle]^{V_i}]^{V_o} \qquad \begin{bmatrix} (V_k, V_g, I) \leftarrow \texttt{groupV}\{\lambda x.x.2\}\ V_i \\ V'_g \qquad\quad \leftarrow \texttt{projectV}\{\lambda x.x.1\}\ V_g \\ V'_k \qquad\quad \leftarrow \texttt{projectV}\{\lambda x.\langle x, \langle\rangle\rangle\}\ V_k \end{bmatrix}}{\Gamma \vdash \texttt{group}^{\uparrow}\ e \Rightarrow [[\langle \delta, [\langle \rho \rangle_I]^{V'_g} \rangle]^{V'_k}]^{V_o}}$$

All aggregate functions are implemented through the same sequence of $\mathcal{SL}$ operators as demonstrated in Figure 33. In Rule SHRED-AGG-LIFT, we map the lifted reduce combinator to the segment folding operator `reduceV{}`.

$$\textsc{shred-agg-lift}\quad \frac{\begin{array}{l}\Gamma \vdash e \Rightarrow [[\tau]^{V_i}]^{V_o} \\ \vdash s_z : \delta_a \end{array} \qquad \begin{bmatrix} V_f \leftarrow \texttt{reduceV}\{s_z, s_f\}\ V_i \\ V_u \leftarrow \texttt{unboxV}\{s_z\}\ V_o\ V_f \\ V \leftarrow \texttt{projectV}\{\lambda x.x.2\}\ V_u \end{bmatrix}}{\Gamma \vdash \texttt{reduce}\{s_z, s_f\}^{\uparrow}\ e \Rightarrow [\delta_a]^{V}}$$

In contrast to other parallel list operations, $\texttt{sng}^{\uparrow}$ and $\texttt{concat}^{\uparrow}$ do not map to segmented $\mathcal{SL}$ operators. These two exclusively modify the list nesting structure. Their vector implementations, accordingly, only modify the segment structure of the vectors involved. Note that neither `segmentV` nor `mergesegV` modify the shape and inner index of a vector. The index relationship with any inner vectors is not affected.

$$\textsc{shred-sng-lift}\quad \frac{\Gamma \vdash e \Rightarrow [\rho]^{V} \qquad \begin{bmatrix} V_o \leftarrow \texttt{projectV}\{\lambda x.\langle\rangle\}\ V \\ V_i \leftarrow \texttt{segmentV}\ V \end{bmatrix}}{\Gamma \vdash \texttt{sng}^{\uparrow}\ e \Rightarrow [[\rho]^{V_i}]^{V_o}}$$

$$\textsc{shred-concat-lift}\quad \frac{\Gamma \vdash e \Rightarrow [[[\rho]^{V_i}]^{V_m}]^{V_o} \qquad [V'_i \leftarrow \texttt{mergesegV}\ V_m\ V_i]}{\Gamma \vdash \texttt{concat}^{\uparrow}\ e \Rightarrow [[\rho]^{V'_i}]^{V_o}}$$

APPENDING LISTS Shredding of the lifted combinator $\texttt{append}^{\uparrow}$ is specified by SHRED-APPEND-LIFT. Primarily, it uses operator `appendV` to append each pair of corresponding segments in $V_{i\cdot 1}$ and $V_{i\cdot 2}$.

$$\textsc{shred-append-lift}\quad \frac{\begin{array}{l}\Gamma \vdash e_1 \Rightarrow [[\rho]^{V_{i\cdot 1}}]^{V_{o\cdot 1}} \\ \Gamma \vdash e_2 \Rightarrow [[\rho']^{V_{i\cdot 2}}]^{V_{o\cdot 2}} \end{array} \qquad [(V_a, I_1, I_2) \leftarrow \texttt{appendV}\ V_{i\cdot 1}\ V_{i\cdot 2}]}{\Gamma \vdash \texttt{append}^{\uparrow}\ e_1\ e_2 \Rightarrow [[\langle\!| \rho |\!\rangle_{I_1} \sqcup \langle\!| \rho' |\!\rangle_{I_2}]^{V_a}]^{V_o}}$$

Two aspects of $\texttt{append}^{\uparrow}$ require special consideration. First, operator `appendV` derives new unique indexes for its result vector V_a: each element of vectors $V_{i\cdot1}$ and $V_{i\cdot2}$ is mapped to a new inner index which is unique in V_a. The mapping of indexes is described by the rekeying transformations I_1 and I_1 which are applied to the respective packages by function $\langle\!| - |\!\rangle_-$. In contrast to other forms of index transformations, however, rekeying transformations do not need to be propagated recursively through the package — a rekeying transformation describes a 1:1-mapping.

$$\begin{aligned}
&\langle\!| \pi |\!\rangle_I = \pi \\
&\langle\!| \langle \ell_i : \rho_i \rangle_{i=1}^n |\!\rangle_I = \langle \ell_i : \langle\!| \rho_i |\!\rangle_I \rangle_{i=1}^n \\
&\langle\!| [\rho]^V |\!\rangle_I = [\rho]^{V'} \\
&\quad \left[V' \leftarrow \texttt{appkeyV}\ I\ V\right]
\end{aligned}$$

Second, it is not sufficient to only apply the top-level vectors. Any vectors in the element packages need to be appended as well. Function $- \sqcup -$ appends two packages with the same structure by recursively appending each pair of vectors.

$$\begin{aligned}
&\pi \sqcup \pi = \pi \\
&\langle \ell_i : \rho_i \rangle_{i=1}^n \sqcup \langle \ell_i : \rho'_i \rangle_{i=1}^n = \langle \ell_i : \rho_i \sqcup \rho'_i \rangle_{i=1}^n \\
&[\rho]^V \sqcup [\rho']^{V'} = [\langle\!| \rho |\!\rangle_{I_1} \sqcup \langle\!| \rho' |\!\rangle_{I_2}]^{V_a} \\
&\quad \left[(V_a, I_1, I_2) \leftarrow \texttt{appendV}\ V\ V'\right]
\end{aligned}$$

If the element packages of branch results merged by $\texttt{combine}^{\uparrow}$ contain inner vectors, these are merged by $- \sqcup -$ as well.

SHRED-COMBINE-LIFT

$$\frac{\begin{array}{l}\Gamma \vdash e_b \wr [[\texttt{Bool}]^{V_b}]^{V_o} \\ \Gamma \vdash e_t \wr [[\rho]^{V_t}]^{V_{o\cdot1}} \\ \Gamma \vdash e_e \wr [[\rho']^{V_e}]^{V_{o\cdot2}} \qquad [V \leftarrow \texttt{combineV}\ V_b\ V_t\ V_e]\end{array}}{\Gamma \vdash \texttt{combine}^{\uparrow}\ e_b\ e_t\ e_e \wr [[\rho \sqcup \rho']^V]^{V_o}}$$

NON-PARALLEL OPERATIONS $\mathcal{FL}$ includes a small set of non-parallel list operators (`restrict`, `combine` and `concat`) that are applied to single list-typed arguments. The flat representation of such a list is an outer vector that contains the unit segment.

Shredding of the former two operations is mostly identical to their data-parallel variants (Rules SHRED-RESTRICT-LIFT and SHRED-COMBINE-LIFT). Both $\mathcal{SL}$ operators `selectV{}` and `combineV` work on the basis of individual vector elements and are oblivious to the segment structure.

SHRED-COMBINE

$$\frac{\begin{array}{l}\Gamma \vdash e_b \wr [\texttt{Bool}]^{V_b} \\ \Gamma \vdash e_t \wr [\rho]^{V_t} \\ \Gamma \vdash e_e \wr [\rho']^{V_e} \qquad [V \leftarrow \texttt{combineV}\ V_b\ V_t\ V_e]\end{array}}{\Gamma \vdash \texttt{combine}\ e_b\ e_t\ e_e \wr [\rho \sqcup \rho']^V}$$

SHRED-RESTRICT

$$\frac{\Gamma \vdash e \wr [\langle \rho, \texttt{Bool} \rangle]^V \qquad \begin{bmatrix}(V_s, I) \leftarrow \texttt{selectV\{}\lambda x.x.2\texttt{\}}\ V \\ V_p \qquad\quad \leftarrow \texttt{projectV\{}\lambda x.x.1\texttt{\}}\ V_s\end{bmatrix}}{\Gamma \vdash \texttt{restrict}\ e \wr [(\!| \rho |\!)_I]^{V_p}}$$

For `concat`, we can define a more efficient implementation than for its lifted counterpart $\texttt{concat}^{\uparrow}$. As the argument is a top-level list, we know that the outer vector V_o consists only of the unit segment. Explicit merging of segments in this case is not necessary. Instead, we simply discard V_o and replace the outer index of V_i with $\langle\rangle$ using `unsegmentV`.

$$\frac{\Gamma \vdash e \Rightarrow [[\rho]^{V_i}]^{V_o} \qquad [V \leftarrow \texttt{unsegmentV}\ V_i]}{\Gamma \vdash \texttt{concat}\ e \Rightarrow [\rho]^{V}}\ \textsc{shred-concat}$$

4.3.3.2 *Running Example Shredded*

As an example for vector programs obtained via shredding, we show the $\mathcal{SL}$ form of the running example Query Q2 in Figure 39. To make it easier to read, we do not show the sequence of $\mathcal{SL}$ statements but use the form of a data-flow plan as in Figure 30. Evaluated via some interpretation of vector programs (*e.g.* $\mathcal{S}[\![-]\!]$), it provides a flat implementation of the nested $\mathcal{CL}$ query.

The $\mathcal{SL}$ program solely describes the runtime aspect of the original $\mathcal{FL}$ expression, *i.e.* the actual computation on vector elements. In particular, the plan shows no traces of the flattening and unflattening of vectors with shape operators. Information about the relationship between vectors is only maintained in the annotations of the result package `[⟨Order,[Lineitem]⟩]`, *i.e.* compile-time meta information.

With lifting and shredding, we obtain comprehensible flat implementations. Due to the compositionality of the translation scheme, the structure of the $\mathcal{SL}$ program can be easily related to the structure of the original $\mathcal{FL}$ expression. The runtime aspect of each $\mathcal{FL}$ operator is implemented as a small group of $\mathcal{SL}$ operators.

The structure of the $\mathcal{SL}$ plan also reflects the structure of iteration scopes in the $\mathcal{SL}$ expression: references to variables that make up the parallel environment manifest in the reuse of vector operator results. Concretely, the plan fragments arising from $\otimes$ and $\texttt{restrict}^{\uparrow}$ represent the $\mathcal{FL}$ expressions bound to variable `l` in the parallel environments — in the original $\mathcal{CL}$ expression, these are the iterator variables `l` of the inner iterators. Any occurrence of `l` in the inner iteration scopes manifests in the DAG-shaped $\mathcal{SL}$ plan as a reference to the corresponding vector operator.

4.3.4 *Optimizing $\mathcal{SL}$*

It is apparent in Figure 39 that scalar computations in $\mathcal{SL}$ programs obtained by shredding are handled rather inefficiently. For example, the computation of the predicate expression $\_.\texttt{ok}^{\uparrow} =^{\uparrow} \_.\texttt{ok}^{\uparrow}$ is split among three `projectV{}` operators. Likewise, pair construction for the outer vector is mapped to an `alignV` operator that essentially aligns `tableV{os}` with itself. These fine-grained patterns are due to the compositional nature of the translation scheme. $\mathcal{FL}$ operators are translated in isolation and each $\mathcal{SL}$ operator implements one specific $\mathcal{FL}$ operator.

On any conceivable backend, directly lowering $\mathcal{SL}$ programs obtained by shredding will lead to inefficient backend code. Every operator results in an intermediate vector. A backend that materializes intermediate results will be constrained by memory bandwidth due to the multitude of intermediate vectors. Problems caused by the excessive materialization of intermedi-

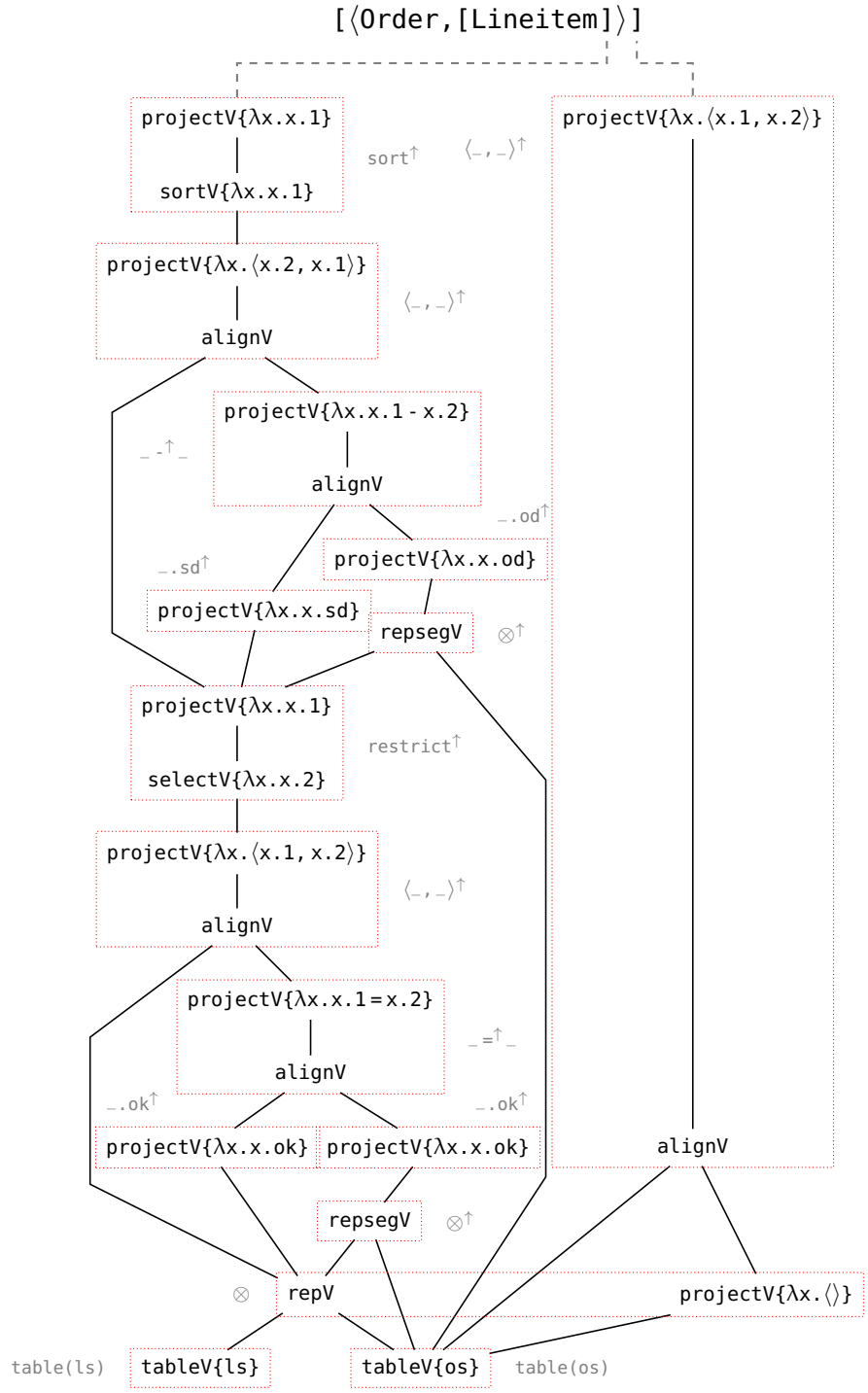


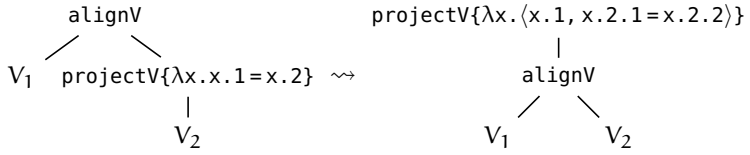
Figure 39: $\mathcal{SL}$ implementation of the running example Query Q2 on flat vectors, derived by shredding. Boxes ▭ mark groups of operators that implement specific $\mathcal{FL}$ operators.

ate results in vectorized programs have been noted both in the domain of data-parallelism [Kel+12; Ber+13] and column-store query engines [BZN05]. On backends that do implement pipelining and avoid the materialization of intermediate results, the size and complexity of plans nevertheless is likely to cause problems: Rittinger [Rit11], for instance, describes problems with large relational algebra plans generated by *Loop-Lifting* (see also Section 2.2.2). The underlying problem is the compositionality of the *Query Flattening* translation scheme that considers each sub-expression independently. Grust *et al.* call this the "compositionality threat" [GMR09].

Instead of dealing with this problem in a backend-specific manner, we reshape $\mathcal{SL}$ programs before lowering them. The goal is a more coarse-grained handling of scalar expressions. A set of straightforward rewrite rules suffices to simplify $\mathcal{SL}$ programs significantly. We sketch the rewrites in DAG plan form on specific patterns in Figure 39. In their general form, rewrites are stated as pattern matches on sequences of statements in an $\mathcal{SL}$ program.

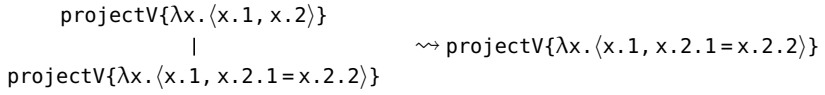
PROJECTION We move `projectV{}` operators towards the top of the plan by pulling them through other operators. The goal is to move `projectV{}` next to other operators to expose inlining opportunities.

- Projections are pulled through `alignV` operators:

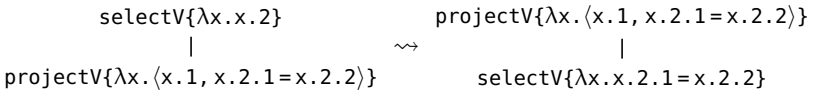


  A symmetric rewrite pulls projections from the left input of an `alignV` operator.

- Adjacent projections are fused:



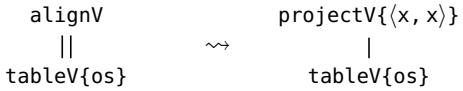
- Projections are pulled through operators like `selectV{}` and `sortV{}`:



Similar rewrites pull `projectV{}` operators through most other $\mathcal{SL}$ operators. The correctness of these rewrites is easily verified based on the semantics of operators in terms of list comprehensions.

ALIGNMENT Pulling projections up fuses scalar computations. Additionally, it clears the way for the elimination of `alignV{}` operators.

- If a vector is aligned with itself, the `alignV` operator is replaced with a projection.



- Stacked `alignV` operators are eliminated if they refer to the same input.

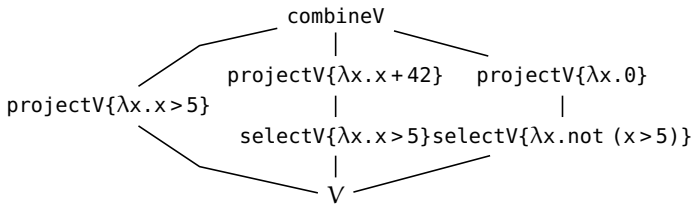
  Variations of this rewrite eliminate all stacked `alignV` operators that refer to the same input.

CONDITIONALS Consider the following $\mathcal{CL}$ expression in which a conditional is iteratively evaluated:

```
[ if x > 5 then x + 42 else 0 | x ← xs ]
```

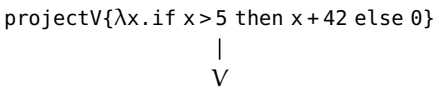
Conditionals are uniformly lifted and shredded into a branch-free form based on the $\mathcal{SL}$ `combineV` operator (Rule LIFT-COND). The general branch-free form is necessary to implement conditionals in which either the condition or a branch expression features complex, list-typed expressions.

In this example, however, all expressions involved are purely scalar. The `projectV{}` and `alignV` rewrites discussed so far will reshape any such scalar conditional into the following form:



Here, we see another instance of non-optimal code due to the compositionality of the translation scheme. Although the plan features only one input vector V, this input is split into three vectors for the condition and both branches. The result of all sub-expressions is explicitly materialized in intermediate vectors. These intermediate vectors have to be merged. Selections and `combineV` have non-trivial runtime cost on any conceivable backend.

Fortunately, this conditional pattern is easy to recognize in a rewrite. Crucially, both selections and the projection in the leftmost `combineV` input feature the same scalar expression (only negated in the rightmost input). $\mathcal{SL}$ offers a much simpler alternative to evaluate scalar conditionals on a vector. Abandoning the branch-free implementation, we rewrite this pattern into the following form



In effect, scalar conditionals are specialized from the general form and evaluated without any intermediate vectors and expensive operators.

PLAN SHAPE With $\mathcal{SL}$ rewrites, we simplify the $\mathcal{SL}$ program for Query Q2 into the form depicted in Figure 40. With respect to scalar computations, this plan behaves much better than the original one. The number of operators and thus the number of intermediate results is considerably smaller.

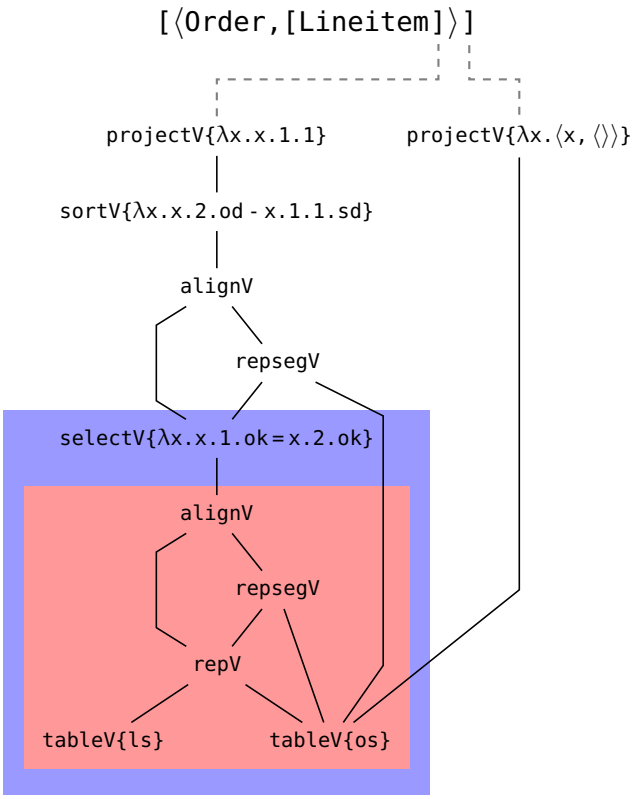


Figure 40: Running example Query Q2 with $\mathcal{SL}$ optimizations applied. The plan region marked red effectively describes the cartesian product of `os` and `ls` while the region marked blue describes their join.

The predicate and sorting keys for `selectV{}` and `sortV{}` are no longer materialized in intermediate vectors.

The optimized $\mathcal{SL}$ plan is no longer obstructed by scalar computations and we can clearly see the iteration structure of the original $\mathcal{CL}$ query. Iteration and variable scopes are directly reflected in the plan. For the inner iteration scopes, operators `repV` and `repsegV` provide the flattened bindings for the parallel variables `o` and `l`, originally bound by iterators. With these two operators, the region marked red in Figure 40 computes all combinations of orders and line items in the required segment structure — the nested-loops semantics of nested iterations turns into data replication. Only afterwards are these combinations restricted to the matching ones by `selectV{}`.

Query Flattening enables a flat comprehension of nested iteration and nested collections. The $\mathcal{SL}$ program in Figure 40 describes the semantics of the nested input query in flat form. Clearly, however, it does not describe a reasonable evaluation strategy. Directly evaluating it leads to a large intermediate result. Only a small portion of it is actually relevant and survives the selection.

Let us disregard for a moment the segment structure necessary to correctly represent the nested query result. The plan region marked red essentially describes the cartesian product of tables `os` and `ls`. Together with the selection (plan region marked blue), it describes an equi-join. Joins are fundamental primitives of query engines, backed by efficient implementations with sorting, hashing or based on indexes [Gra93]. Merging cartesian products and selections into joins is a standard measure of any logical query optimizer [JK84].

Once we lower the $\mathcal{SL}$ program to backend code, however, a backend optimizer will have a hard time introducing a join. The plan does not fit the standard σ-$\times$-patterns expected by query optimizers. Additionally, backend code will be further complicated by having to maintain the segment structure and — for an unordered backend — order information.

On a slightly higher level of abstraction, we can introduce additional $\mathcal{SL}$ rewrites that recognize certain patterns of $\mathcal{SL}$ operators and rewrite them into specialized operators. Arbitrarily nested lists and nesting of iterators allowed in the orthogonal language $\mathcal{CL}$ lead to a considerable diversity in the $\mathcal{SL}$ patterns that have to be considered. Furthermore, $\mathcal{SL}$ rewrites critically have to maintain segment structure. As a consequence, an $\mathcal{SL}$ optimizer that fundamentally reshapes the structure of $\mathcal{SL}$ queries requires a complicated rule set. In our work, relying on $\mathcal{SL}$ rewrites alone to obtain good backend queries turned out not to be a viable option.

Instead, we take a more high-level approach in Chapter 5. There is no reason to change the query structure into more efficient forms only after flattening. Optimization and deriving a flat implementation are actually orthogonal. As we are mainly interested in specializing the iteration structure into more efficient forms, we optimize $\mathcal{CL}$ queries before flattening. Here, the iteration structure is most clearly available.

4.4 EXTENSIBILITY

In this chapter we have described *Query Flattening* for $\mathcal{CL}$ with a fixed set of built-in list combinators. However, *Query Flattening* can be easily extended with new list combinators. Note that lifting treats all built-in combinators uniformly with Rule LIFT-BUILTIN. No changes to lifting are necessary to support an additional combinator p as long as we can supply a data-parallel

version $p^{\uparrow}$ of the combinator. To integrate $p^{\uparrow}$ with shredding and implement it on segment vectors, we only have to provide one additional shredding rule.

As an example, consider adding a `scan{}` combinator to $\mathcal{CL}$ that has the following typing rule and $\mathcal{ML}$ interpretation:

$$\frac{\Gamma \vdash e : [\tau] \qquad \vdash s_z : \delta_a \qquad \vdash s_f : \delta_a \to \tau \to \delta_a}{\Gamma \vdash \texttt{scan\{}s_z\texttt{, }s_f\texttt{\}}\ e : [\delta_a]} \ \mathcal{CL}\text{-TY-SCAN}$$

$$\mathcal{I}[\![\texttt{scan\{}s_z\texttt{, }s_f\texttt{\}}\ e]\!]\rho = \mathsf{scan}\ [\![s_f]\!]\ [\![s_z]\!]\ \mathcal{I}[\![e]\!]\rho \qquad (\mathcal{CL}_d\text{-SCAN})$$

The `scan{}` combinator provides running aggregates of lists. The `mins` running minimum of Section 1.3.1, for instance, can be expressed as

```
scan{∞, λm x.min m x}
```

As a side note, we point out that the scan operator plays a significant role in Blelloch's discussion of parallel algorithms [Ble90].

We omit the trivial definition of the lifted $\texttt{scan\{}\texttt{\}}^{\uparrow}$ combinator. Shredding occurrences of $\texttt{scan\{}\texttt{\}}^{\uparrow}$ requires the definition of a segment scan operator in $\mathcal{SL}$:

$$\frac{\Gamma \vdash V : \mathsf{D}\ (\alpha, \beta, \gamma) \qquad \vdash s_z : \delta \qquad \vdash s_f : \delta \to \gamma \to \delta}{\Gamma \vdash \texttt{scanV\{}s_z\texttt{, }s_f\texttt{\}}\ V : \mathsf{D}\ (\alpha, \beta, \delta)} \ \mathcal{SL}\text{-TY-SCAN}$$

$$\mathcal{S}[\![\texttt{scanV\{}s_z\texttt{, }s_f\texttt{\}}\ V]\!]\rho = \\ \mathsf{concat}\ [\ \mathsf{scan}\ ([\![s_f]\!] \oslash \pi_p)\ [\![s_z]\!]\ \mathsf{seg.2} \mid \mathsf{seg} \leftarrow \mathsf{segs}\ \rho(V)\ ] \qquad (\mathcal{SL}\text{-DISTINCT})$$

Given this definition, the shredding rule is defined easily:

$$\frac{\Gamma \vdash e \wr [[\tau]^{V_i}]^{V_o} \qquad \vdash s_z : \delta_a \qquad [V \leftarrow \texttt{scanV\{}s_z\texttt{, }s_f\texttt{\}}\ V_i]}{\Gamma \vdash \texttt{scan\{}s_z\texttt{, }s_f\texttt{\}}^{\uparrow}\ e \wr [\delta_a]^{V}} \ \text{SHRED-SCAN-LIFT}$$

In the relational world, `scanV{}` can be implemented with partitioned window aggregates. In Chapter 5 we extend $\mathcal{CL}$ with further combinators for optimization of comprehensions.

4.5 RELATED WORK

Query Flattening directly implements the core concepts of the flattening transformation as originally described by Blelloch and Sabot [BS89]. We limit *Query Flattening* to a language without functions and handle only expressions, not function definitions. This is not a fundamental restriction, though. The flattening transformation for first-order languages generates lifted versions of user-defined functions by lifting the function body expression — this would be easy to add. Only *higher-order flattening* described by Leshchinskiy [Les05] takes a substantially different route. As explained in Section 1.4.2, we think that the additional complications for higher-order flattening are not warranted in the context of query flattening.

Central to *lifting* is the handling of nested iterators and the corresponding nested variable scopes. Lifting is usually implemented with rewrite rules

that eliminate iterators by pushing them through other expressions. The following rules are the core of the transformation and can be found verbatim in the works of Palmer and Prins [PP95], Keller and Simons [KS96] and Keller [Kel99]:

$$[\ f^n\ e_1\ \cdots\ e_n\ |\ x \leftarrow xs\ ] = f^{n+1}\ [\ e_1\ |\ x \leftarrow xs\ ]\ \cdots\ [\ e_n\ |\ x \leftarrow xs\ ] \tag{1}$$

$$[\ x\ |\ x \leftarrow xs\ ] = xs \tag{2}$$

$$[\ y\ |\ x \leftarrow xs\ ] = y \otimes xs \quad x \neq y \tag{3}$$

Equation (1) pushes an iterator through a function application by increasing the lifting level (initially 0) of the function. Equation (2) replaces an occurrence of the iterator variable with the generator expression. Equation (3) distributes a variable that is not the iterator variable. Constants are distributed in the same fashion as Equation (3). Starting with the innermost iterator, these rules eliminate one iterator at a time. Note that repeated pushing of iterators leads to lifted occurrences of $\otimes$. For nested iterators, lifting in this fashion leads to deeply nested chains of $\otimes^{\uparrow}$ that replicate data through each individual nesting level step by step. The resulting code is not only inefficient but also hard to comprehend.

Our description of lifting is different in that it does not eliminate variable bindings completely upfront. It maintains nested variable scopes but replaces variable bindings due to iterators with the explicit construction of a *parallel environment* (Rule LIFT-ITER). Constants are not replicated level by level. Instead, we track the nesting depth d explicitly and use $\divideontimes_d$ to directly replicate at the correct nesting level. Hence, replication of constants does not require $\otimes^{\uparrow}$ and the resulting lifted expressions are considerably more concise. An optimization with the same effect is described by Keller and Simons [KS96] as a separate post-mortem rewrite rule. Madsen [Mad16] describes a version of lifting very similar to ours.

The description of flattening usually targets physically ordered vectors and is tied to length and positions of elements. Lifted operations are implemented based on element positions or using inherently orderered combinators like `zip`. In contrast, we define flattening to rely exclusively on identity of list elements provided by indexes. This prepares for the lowering to unordered backends where we aim to minimize any effort for order maintenance.

5 QUERY FLATTENING AND QUERY OPTIMIZATION

Query Flattening as described in Chapter 4 enables a flat comprehension of nested queries and nested data. The resulting flat $\mathcal{SL}$ queries encode an inefficient evaluation strategy, though (Section 4.3.4). Algebraic $\mathcal{FL}$ queries implement nested iteration by replicating data. Base tables that occur in an iteration scope are replicated to provide their value in each iteration independently. Variable bindings are propagated to inner iteration scopes by replicating their values (environment lifting). Effectively, *Query Flattening* takes the nested-loop semantics of nested iterators literally. The resulting flat queries suffer from large intermediate results and a complex structure. Deriving idiomatic and efficient relational queries is difficult.

Nested iteration, however, is prevalent in queries. In particular, nested comprehensions that are correlated by predicates as in Query Q1 — *i.e. correlated* subqueries — occur in the vast majority of queries. We are not only interested in a flat comprehension of queries, but in a practical and efficient one. Hence, the ability to generate efficient backend code for typical query patterns is paramount.

In this chapter, we show that well-established methods from logical query optimization are sufficient to eliminate replication due to flattening. Contrary to prior work by Rittinger [Rit11], we don't rely on purely relational rewrites to reshape flat queries. Instead, we focus on the optimization of comprehensions in the original $\mathcal{CL}$ queries (Section 5.2). Our main result in Sections 5.3 and 5.4 is to show that these methods are orthogonal to query flattening and integrate well with the framework of lifted combinators and flat data representation we set up in Chapter 4.

5.1 AVOIDING REPLICATION IN FLATTENING

Problems due to replication of data are not specific to *Query Flattening*. All implementations of Blelloch's flattening transformation (Chapter 3) suffer from replication and the resulting large intermediate results [Ble95; PP95; Lip+12; Pal+95]. Blelloch, for instance, notes that iterative positional indexing into a list requires a copy of the list for each iteration [Ble95, Appendix C]. *Loop-Lifting* — basically the flattening translation phrased in relational algebra (Section 2.2.2) — suffers from the same problems [GMR09; Rit11].

A number of proposed solutions to these problems focus on positional indexing [PP95; KS96], but do not cover correlated iteration. More general approaches refrain from fully set-oriented evaluation [Pal+95] or share segments physically with pointer-based representations of segment vectors [RP00; Lip+12]. These approaches, however, assume a particular parallel runtime and do not match our setting of generating code for general relational engines.

Specific to query flattening, a number of authors propose to rewrite low-level relational algebra expressions produced by *Loop-Lifting* [Rit11; Gru05]. Those relational expressions encode replication and are additionally obscured by the maintenance of list order and nested data (Section 2.2.2). Rewriting those complex plans requires an equally complex set of rules as

well as an intricate rewriting strategy and has been shown to fail in simple cases (Section 2.2.2).

We have described query flattening as a sequence of lowerings that peel of layers of abstraction. All optimizations that we might want to perform can in principle be expressed at the very last stage of lowering, *i.e.* in the most low-level language. However, at this point, most optimizations will not be *convenient* to express because the problematic patterns we would like to optimize are blurred and hard to recognize. Instead, individual optimizations should be performed at the specific level of abstraction at which problematic patterns can be observed first and the corresponding optimization can be expressed best [Sha+16].

Is the problem of data replication forced by the flat data and operations in $\mathcal{SL}$? In our two-phase approach in which lifting precedes shredding, data replication is introduced in the lifting phase. The $\mathcal{FL}$ queries resulting from $\mathcal{L}[\![-]\!]$ encode data replication in the form of $\otimes$ and $\otimes^{\uparrow}$ operators. Hence, we can focus on the lifting phase that maps the calculus-based language $\mathcal{CL}$ to the algebraic language $\mathcal{FL}$. Every description of the flattening transformation includes a translation to an algebraic form without variable bindings (Chapter 3) to enable *set-oriented* or *data-parallel* evaluation.

Are problems with data replication specific to the flattening transformation? Actually, they are well-known from the implementation of query calculi: a naive translation from a query calculus with nested queries to an algebra enforces a nested-loops evaluation strategy [Cod72; Ste95; Suc97; RKS88]. Hence, we argue that the major problems in flattened queries do not require flattening-specific optimization techniques. Essentially, we have to derive efficient algebraic $\mathcal{FL}$ forms from $\mathcal{CL}$ calculus queries prior to shredding that do not encode replication.

Deriving efficient algebraic forms from calculus queries has been extensively researched in the field of logical query optimization — both for relational [JK84] and complex-object models [Ste95]. In this chapter, we adopt this work to derive efficient algebraic $\mathcal{FL}$ expressions from $\mathcal{CL}$ queries. The optimization techniques we employ are neither novel nor specific to query flattening. We rely on prior work on logical query optimization for complex-object query languages [Ste95; Gru99; GS99; CM93; Cla+97; FM00]. In particular, we closely follow the approach outlined by Grust [Gru99], and Grust and Scholl [GS99].

5.2 OPTIMIZING ITERATIONS

Steenhagen [Ste95] summarizes two approaches to translate calculus queries into efficient expressions in an algebraic intermediate representation:

- Efficient algebraic expressions can be obtained by translating into the algebra in a naive or canonical way first and rewriting into more efficient equivalent expressions afterwards [JK84].
- Alternatively, optimization can be included into the translation to the algebra [Ste95; Nak90; FM00].

In our framework, these alternatives amount to either (a) rewriting $\mathcal{FL}$ expressions that result from the lifting transformation $\mathcal{L}[\![-]\!]$, or (b) extending $\mathcal{L}[\![-]\!]$ with additional rules. Both alternatives lead to an increase in complexity of individual translation steps. Alternative (a) would require an involved system of algebraic rewrites that can deal with the diversity of $\mathcal{FL}$ expressions caused by the compositional translation scheme. Alternative (b),

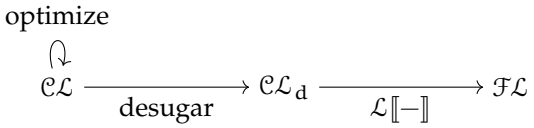


Figure 41: Optimization of $\mathcal{CL}$ queries prior to *Query Flattening*.

on the other hand, would increase the complexity of $\mathcal{L}[\![-]\!]$ considerably. Hence, we prefer to consider optimization separately.

Although query optimization is traditionally performed on an algebraic representation, this is not a necessity. Grust and Scholl [GS99] argue that comprehension calculi are a useful intermediate representation for queries and provide a good starting point for logical query optimization. Comprehensions can be rewritten into a normal form in which typical iteration patterns have a canonical representation and are easy to recognize. We adopt their point of view and optimize queries in the comprehension calculus $\mathcal{CL}$ before translating to the algebra $\mathcal{FL}$ (Figure 41). This allows us to keep optimization and translation separate and simple.

In the following, we assemble the ingredients of a rule-based $\mathcal{CL}$ optimizer. We rewrite $\mathcal{CL}$ queries into equivalent queries such that replication can be avoided during lifting. Rewrite rules simplify the iteration structure of $\mathcal{CL}$ queries and introduce specialized operators that encapsulate certain iteration patterns.

5.2.1 *Rewrite Notation*

We specify rewrite rules on $\mathcal{CL}$ expressions in the notation introduced in Section 4.1. In some rewrite rules, we refer to the location of a sub-expression relative to its enclosing expression. We write

$$e \setminus\!\!\downarrow e_1$$

to refer to a sub-expression of e that matches the syntactic form of e_1. More precisely, we let $e \setminus\!\!\downarrow e_1$ denote the first occurrence of an expression that matches e_1 in a pre-order traversal of the abstract syntax tree representation of e. We restrict this traversal in one regard: we do not let it descend into the head expression of nested comprehensions.

If the left-hand side of a rule features a sub-expression pattern, we use the same syntax on the right-hand side to replace the sub-expression. We write

$$e \setminus\!\!\downarrow e_1 \rightsquigarrow e \setminus\!\!\downarrow e_2$$

to indicate that in expression e sub-expression e_1 is replaced by e_2.

5.2.2 *Normalizing $\mathcal{CL}$ Expressions*

Our primary objective is to avoid replication of data due to nested iteration. As a preparation, we utilize a set of well-known rewrite rules that eliminate redundant computation in $\mathcal{CL}$ queries. We only sketch these rules here and focus on comprehension-specific aspects. We refer to Wong [Won94] for a extensive discussion.

As a first measure, we employ standard partial evaluation techniques. This includes constant folding for scalar operations, fusing record creation

and record field selection as well as eliminating conditionals if the condition is constant. We eliminate bindings `let` x `=` e_1 `in` e_2 by inlining e_1 into e_2 if x is referenced not more than once in e_2. However, we never inline into the head of nested comprehensions to avoid an unnecessary iterative evaluation of e_1 (see also Section 5.2.3). We evaluate list computations over constant lists at compile time, thus for instance propagating empty lists. Finally, we move `let`-bindings as much as possible towards the root of the expression.

Redundant comprehensions are removed by the following Rule NORM-ID.

$$[\ x \mid x \leftarrow xs\ ] \rightsquigarrow xs \qquad \text{(NORM-ID)}$$

We employ a number of comprehension-specific rewrites to eliminate intermediate list nesting whenever possible. The following rules fuse the creation of nested lists with subsequent list flattening by `concat`.

$$\texttt{concat}\ [\ [\ e \mid qs'\ ] \mid qs\ ] \rightsquigarrow [\ e \mid qs, qs'\ ] \qquad \text{(NORM-CONCAT)}$$

$$\texttt{concat}\ [\ \texttt{sng}\ e \mid qs\ ] \rightsquigarrow [\ e \mid qs\ ] \qquad \text{(NORM-CONCAT-SNG)}$$

$$\texttt{concat}\ [\ \texttt{if}\ e_1\ \texttt{then sng}\ e_2\ \texttt{else []} \mid qs\ ] \rightsquigarrow [\ e_2 \mid qs, e_1\ ] \qquad \text{(NORM-CONCAT-IF)}$$

$$\texttt{concat}\ [\ \texttt{if}\ e\ \texttt{then}\ [\ h \mid qs'\ ]\ \texttt{else []} \mid qs\ ] \rightsquigarrow [\ h \mid qs, e, qs'\ ] \qquad \text{(NORM-CONCAT-IF-COMP)}$$

The effect of Rules NORM-CONCAT to NORM-CONCAT-IF-COMP opposes desugaring with Rule DESUGAR-GENS (Section 4.1). Indeed, these rules can be considered a form of *resugaring* [Ale+15] that transforms deeply nested iterators of the simple form $[\ e_1 \mid x \leftarrow e_2\ ]$ into a canonical form of comprehensions with multiple generators and guards. In particular, resugaring recovers canonical comprehensions from $\mathcal{CL}$ queries obtained by a compositional translation of a source language (Section 1.4.2). We employ resugaring and desugaring in different phases of the compiler and thus avoid any conflicts. We resugar to obtain a canonical representation during optimization and desugar to iterators only afterwards (Figure 41).

The orthogonality of $\mathcal{CL}$ allows arbitrary nesting of comprehensions. As the iteration structure is directly reflected in flattened queries, it pays off to eliminate redundant comprehensions. We simplify comprehensions with a set of well-known *comprehension normalization* [Gru99; FM00] rules. Three rules simplify comprehensions due to specific forms of generator expressions. Rules NORM-EMPTY and NORM-SNG handle empty and singleton generators (a variant of Rule NORM-SNG deals with singleton literal lists).

$$[\ e \mid qs, x \leftarrow [\,], qs'\ ] \rightsquigarrow [\,] \qquad \text{(NORM-EMPTY)}$$

$$[\ e \mid qs, x \leftarrow \texttt{sng}\ e', qs'\ ] \rightsquigarrow [\ e[e'/x] \mid qs, qs'[e'/x]\ ] \qquad \text{(NORM-SNG)}$$

Most importantly among the normalization rules, Rule NORM-GEN inlines comprehensions nested in a generator expression.

$$[\ e \mid qs, x \leftarrow [\ e' \mid qs''\ ], qs'\ ] \rightsquigarrow [\ e[e'/x] \mid qs, qs'', qs'[e'/x]\ ] \qquad \text{(NORM-GEN)}$$

Two technicalities have to be observed for a correct implementation of Rules NORM-GEN and NORM-SNG:

- Substitution in qualifiers qs' may require renaming to avoid capturing free variables in e'. We refer to our remarks in Section 4.1. To simplify the exposition, we assume that $fv(e')$ and $bv(qs')$ are disjunct and thus no renaming is necessary.

- As qualifiers $\mathtt{qs}''$ are lifted to the outer comprehension, variables that occur free in $\mathtt{qs}'$ and e must not be shadowed. Here, we assume that $bv(\mathtt{qs}'')$ and $fv(\mathtt{qs}') \cup fv(e)$ are disjunct.

The normalizing rewrites described in this section reduce the variability of expression in $\mathcal{CL}$ queries and thus ease the task of subsequent optimizations. At the same time, these rewrites eliminate redundant computations and are thus reasonable heuristic optimizations on their own. Consider the following application of Rule NORM-GEN:

```
[ ⟨x, z⟩ | x ← xs, z ← [ y | y ← ys, p x y ] ]
≡  { NORM-GEN }
   [ ⟨x, y⟩ | x ← xs, y ← ys, p x y ]
```

The rewritten variant is clearly preferable. Unnesting the inner comprehension fuses two loops and eliminates a redundant intermediate result that otherwise might have to be materialized. More importantly though, normalization combines the predicate and both generators in one comprehension and allows subsequent rewrites to introduce a join combinator (Section 5.2.4).

Inlining comprehensions with Rule NORM-GEN reduces the number of iteration scopes and thereby the effort for environment lifting. Consider the effects of lifting on the following example:

```
[ [ ⟨x, y⟩ | y ← [ ⟨x, z⟩ | z ← zs ] ] | x ← xs ]                (Q6)
≡  { NORM-GEN }
   [ [ ⟨x, ⟨x, z⟩⟩ | z ← zs ] | x ← xs ]
```

The free occurrence of variable x in both inner iteration scopes leads to environment lifting for both scopes and thus two occurrences of $\otimes^{\uparrow}$ (Rule LIFT-ITER). In the rewritten expression, both inner comprehensions are fused and x has to be lifted only once.

5.2.3 *Loop-Invariant Expressions*

Simple techniques suffice to eliminate replication of data in certain cases. Consider Query Q7 in which the guard expression `sum ys` does not depend on the comprehension variable `x` (let `ys` be a list constant).

```
[ x | x ← xs, x > (sum ys) ]                                     (Q7)
≡  { DESUGAR-PRED, DESUGAR-TOP }
   concat [ restrict [ ⟨x, x > (sum ys)⟩ | x ← xs ] | z ← [⟨⟩] ]
```

Lifting results in an $\mathcal{FL}$ expression that replicates `ys` as follows:

$$[\![\, \mathtt{sum\ (ys} \ltimes_2 \mathtt{(xs} \ltimes_1 \mathtt{[\langle\rangle]))} \,]\!]_2$$

This term clearly does not encode a reasonable evaluation strategy: For each element of `xs`, an independent copy of `ys` is created, potentially resulting in a large intermediate result. Evaluating $\mathtt{sum}^{\uparrow}$ on these copies duplicates work.

As sum ys does not depend on x, there is no actual reason to evaluate it iteratively. The following rewrite rules factor expressions out of comprehensions if none of the variables bound by generators in qs occur free:

$$[\ h \searrow e \mid qs\ ] \rightsquigarrow \texttt{let}\ v = e\ \texttt{in}\ [\ h \searrow v \mid qs\ ] \qquad \text{(FACTOR-HEAD)}$$

$$[\ h \mid qs, p \searrow e, qs'\ ] \rightsquigarrow \texttt{let}\ v = e\ \texttt{in}\ [\ h \mid qs, p \searrow v, qs'\ ] \qquad \text{(FACTOR-GUARD)}$$

$$[\ h \mid qs, x \leftarrow e \searrow e', qs'\ ] \rightsquigarrow \texttt{let}\ v = e'\ \texttt{in}\ [\ h \mid qs, x \leftarrow e \searrow v, qs'\ ] \qquad \text{(FACTOR-GEN)}$$

We understand $h \searrow e$ to denote the maximal sub-expression of h encountered in a pre-order traversal of h, such that $fv(e)$ and $bv(qs)$ are disjunct. Again, we do not let the traversal descend into nested comprehensions.

With Rule FACTOR-GUARD in place, we can rewrite Query Q7 as follows:

```
[ x | x ← xs, x > (sum ys) ]
≡   { FACTOR-GUARD }
  let s = sum ys in [ x | x ← xs, x > s ]
≡   { DESUGAR-PRED, DESUGAR-TOP }
  concat [ let s = sum ys in restrict [ ⟨x, x > s⟩ | x ← xs ]
         | z ← [⟨⟩] ]
```

Here, the constant expression sum ys is evaluated outside of the scope of the inner iterator. Consequently, the sum is computed only for a single copy of ys:

$$\llbracket\ \texttt{sum}\ (\texttt{ys} \ltimes_1 [\langle\rangle])\ \rrbracket_1$$

Subsequently, environment lifting for the innermost iteration scope propagates the singleton result of $\texttt{sum}^{\uparrow}$ over all elements of xs.

Note that factoring the list ys alone would not improve the situation for flattening. Since the constant computation on ys (sum ys) would still be evaluated iteratively, lifting would replicate the data nevertheless. To actually improve the situation, the computation itself has to be factored out. Hence, we restrict factoring to complex expressions — *i.e.* expressions involving list combinators and comprehensions.

Factoring expressions out of iterations is profitable even if we can't move them to the top level. Consider Query Q8 in which a complex predicate is correlated with the outer but not the inner comprehension.

```
[ [ y | y ← ys, y > (sum [ z | z ← zs, x = z ]) ] | x ← xs ]        (Q8)
≡   { FACTOR-GUARD }
  [ let u = sum [ z | z ← zs, x = z ] in [ y | y ← ys, y > u ]
  | x ← xs ]
```

Lifting leads to the following replication patterns for Query Q8 and the rewritten version, respectively.

$$\llbracket\ \texttt{sum}\ (\texttt{zs} \ltimes_2 (\texttt{ys} \ltimes_1 \texttt{xs}))\ \rrbracket_2 \qquad \text{(original)}$$

$$\llbracket\ \llbracket\ \texttt{sum}\ (\texttt{zs} \ltimes_1 \texttt{xs})\ \rrbracket_1 \otimes (\texttt{ys} \ltimes_1 \texttt{xs})\ \rrbracket_1 \qquad \text{(rewritten)}$$

Clearly, the latter version is an improvement: zs is only replicated for elements of xs, not all combinations of ys and xs. More importantly, this

$$\texttt{thetajoin\{s\}}\ e_1\ e_2 = \texttt{[⟨x, y⟩ | x ← } e_1\texttt{, y ← } e_2\texttt{, s x y]} \quad \text{(THETAJOIN)}$$

$$\texttt{semijoin\{s\}}\ e_1\ e_2 = \texttt{[x | x ← } e_1\texttt{, or [s x y | y ← } e_2\texttt{]]} \quad \text{(SEMIJOIN)}$$

$$\texttt{antijoin\{s\}}\ e_1\ e_2 = \texttt{[x | x ← } e_1\texttt{, and [¬(s x y) | y ← } e_2\texttt{]]} \quad \text{(ANTIJOIN)}$$

$$\texttt{nestjoin\{s\}}\ e_1\ e_2 = \texttt{[⟨x, [⟨x, y⟩ | y ← } e_2\texttt{, s x y]⟩ | x ← } e_1\texttt{]} \quad \text{(NESTJOIN)}$$

Figure 42: Definition of $\mathcal{CL}$ join combinators, expressed in terms of $\mathcal{CL}$.

rewrite brings the innermost comprehension closer to the outer comprehension with which it is correlated and thus enables other rewrites (Section 5.2.4).

5.2.4 *Introducing Join Combinators*

The techniques introduced so far eliminate redundant nesting of data and iteration and avoid the iterative evaluation of constant expressions. However, for a large class of queries, exhaustively applying these rewrites does not avoid replication. Consider our running example Query Q1:

```
[ ⟨o, sort [ ⟨l, l.sd - o.od⟩ | l ← ls, l.ok = o.ok ]⟩
| o ← os
, 5 < sum [ l.sd - o.od | l ← ls, l.ok = o.ok ] ]
```

This query is in normal form — comprehension normalization rules do not apply. Furthermore, none of the expressions evaluated iteratively is constant. Lowering via $\mathcal{L}[\![-]\!]$ leads to backend code that replicates `ls` and evaluates the predicate on each copy. All queries that feature nested iteration with independent generator expressions (here: table references `os`, `ls`) correlated by a predicate will end up in the same situation.

Centered around comprehensions, $\mathcal{CL}$ can not express this pattern in a different way that would avoid replication. Hence, we follow Grust [Gru99] and extend $\mathcal{CL}$ with *join combinators* that encapsulate specific forms of nested iteration. We list all join combinators in Figure 42 and their typing rules in Figure 43. Being defined in $\mathcal{CL}$ themselves, join combinators clearly do not increase the expressiveness of $\mathcal{CL}$. By rewriting specific iteration patterns into join combinators, we prevent $\mathcal{L}[\![-]\!]$ from introducing replication. Join combinators can be directly lowered to equivalent operators in $\mathcal{FL}$ without falling back to the canonical implementation of nested iteration. To $\mathcal{L}[\![-]\!]$, join combinators appear as blackboxes, just as other list combinators like `sort` and `group`.

All join combinators in Figure 42 are well-known: `thetajoin{}`, `semijoin{}` and `antijoin{}` are standard list-based equivalents of the usual relational join operators $\bowtie$, $\ltimes$ and $\triangleright$. The `nestjoin{}` combinator has been introduced by Steenhagen *et al.* [SAB94] to unnest complex-object queries. It covers nested iteration in the head of a comprehension as well as in qualifiers. For each element of a list `xs`, `nestjoin{s} xs ys` computes the list of matching elements in list `ys`. These inner lists may be empty, indicating the absence of matching elements.

Similar to the folding combinator `reduce{z, f}`, join combinators are parameterized with *scalar* functions that specify join predicates. With the predi-

$\mathcal{CL}$-TY-THETAJOIN

$$\frac{\Gamma \vdash e_1 : [\tau_1] \qquad \Gamma \vdash e_2 : [\tau_2] \qquad \vdash s : \tau_1 \to \tau_2 \to \texttt{Bool}}{\Gamma \vdash \texttt{thetajoin\{s\}}\ e_1\ e_2 : [\langle \tau_1, \tau_2 \rangle]}$$

$\mathcal{CL}$-TY-QUANTJOIN

$$\frac{\Gamma \vdash e_1 : [\tau_1] \qquad \Gamma \vdash e_2 : [\tau_2] \qquad \vdash s : \tau_1 \to \tau_2 \to \texttt{Bool}}{\Gamma \vdash \texttt{semijoin/antijoin\{s\}}\ e_1\ e_2 : [\tau_1]}$$

$\mathcal{CL}$-TY-NESTJOIN

$$\frac{\Gamma \vdash e_1 : [\tau_1] \qquad \Gamma \vdash e_2 : [\tau_2] \qquad \vdash s : \tau_1 \to \tau_2 \to \texttt{Bool}}{\Gamma \vdash \texttt{nestjoin\{s\}}\ e_1\ e_2 : [\langle \tau_1, [\langle \tau_1, \tau_2 \rangle] \rangle]}$$

Figure 43: Typing rules for $\mathcal{CL}$ join combinators.

cate fixed, join combinators are regular list combinators with two arguments. We deliberately support only scalar predicates to preserve the uniformity of query flattening. Subsequent lowering stages can map scalar predicates directly to relational queries. Complex predicates that involve list computations, on the other hand, need to be expressed by list combinators and comprehensions and are thus subject to flattening. Considering join predicates particularly next to the regular lowering pipeline of query flattening is not necessary. Join combinators enable us to map such complex predicates to idiomatic backend queries.

To demonstrate how join combinators eliminate replication, we use the `nestjoin{}` combinator to rewrite Query Q1 into the following Query Q9:

```
[ ⟨u.1, sort [ ⟨l, v.2.sd - v.1.od⟩ | v ← u.2 ]⟩
| u ← nestjoin{λo l.l.ok = o.ok} os ls                    (Q9)
, 5 < sum [ v.2.sd - v.1.od | v ← u.2 ] ]
```

In Query Q9, `nestjoin{}` encapsulates the heavy lifting of computing matching elements of table `ls` for each element of `os`. Both `os` and `ls` are arguments to the same list combinator and occur in the same outer iteration scope. Hence, lifting will not replicate `ls` over `os`. The nested comprehensions that remain in Query Q9 apply scalar operations to elements of the nested lists produced by `nestjoin{}`. This is already handled well by $\mathcal{L}[\![-]\!]$: those inner comprehensions translate to lifted combinators (*i.e.* $\texttt{-}^{\uparrow}$, $\langle -, - \rangle^{\uparrow}$) and zero-cost flattening and unflattening of nested lists ($\texttt{forget}_n$, $\texttt{imprint}_n$).

Most join combinators are close to relational operators. Still, there is a considerable gap to bridge from list-based $\mathcal{CL}$ joins to relational joins. Any implementation of a $\mathcal{CL}$ join combinator has to preserve the semantics of its defining list comprehension faithfully: join combinators preserve the list order of inputs and accept lists whose elements are arbitrary combinations of records and nested lists. Furthermore, the `nestjoin{}` combinator produces a nested result and has no obvious relational counterpart.

We have described query flattening as a sequence of lowerings that peel of certain aspects (nested iteration, nested lists, order) on the way to flat and eventually relational queries. Relational engines have particularly efficient implementations of join operators that we should target. Hence, we peel off all non-relational aspects but map the core aspect of the combinator — the correlated iteration pattern that it expresses — to its relational counterpart.

In the remainder of this chapter, we demonstrate that lowering of join combinators indeed integrates well with the concepts introduced in Chapter 4. We end up with idiomatic and efficient flat queries.

5.2.4.1 *Avoiding Environment Lifting*

As in the work of Steenhagen *et al.* [SAB94], the result of `nestjoin{}` pairs each element of the first argument with the group of matching elements of the second argument. Steenhagen *et al.* originally define the nestjoin operator as follows (in our notation):

$$\bigwedge \{s\}\ e_1\ e_2 = [\ \langle x, [\ y \mid y \leftarrow e_2, s\ x\ y\ ]\rangle \mid x \leftarrow e_1\ ]$$

With (NESTJOIN) we slightly modify their definition for our combinator `nestjoin{}`: groups additionally contain the corresponding element of the first argument list in every element. This seemingly trivial change has proven to be of considerable consequence in translating nested comprehensions: in many cases, it enables us to eliminate variable references across iteration scopes and thus the need for environment lifting (Rule LIFT-ITER). This simplifies the structure of queries considerably and avoids runtime cost for the $\otimes^{\uparrow}$ operator.

Concretely, in Query Q1 the head of the inner comprehensions contains a free occurrence of variable `o` that is bound by the outer comprehension. In Query Q10, on the other hand, we replace `o` with `v.1`, *i.e.* references to the current group element. As a consequence, Query Q10 avoids environment lifting from the get-go.

```
[ ⟨u.1, sort [ ⟨l, v.2.sd - v.1.od⟩ | v ← u.2 ]⟩
| u ← nestjoin{λo l.l.ok = o.ok} os ls                    (Q10)
, 5 < sum [ v.2.sd - v.1.od | v ← u.2 ] ]
```

5.2.4.2 *Introduction Rules for Join Combinators*

Any $\mathcal{CL}$ expression that matches the right-hand side of an equation in Figure 42 can be replaced by the corresponding combinator. Exact matches are rare, however and only capture a small subset of queries. In addition, we side each join operator with one or multiple *introduction rules*: rewrite rules that match patterns in comprehensions and replace them with a join combinator. Rule ANTIJOIN-12, for instance, introduces `antijoin{}` for a pattern that expresses universal quantification with a *range* predicate p_r and a *quantifier* predicate p_q.

$$[\ h \mid qs, x \leftarrow xs, \texttt{and}\ [\ p_q\ x\ y \mid y \leftarrow ys, p_r\ y\ ], qs'\ ]$$
$$\rightsquigarrow$$
$$[\ h \mid qs, x \leftarrow \texttt{antijoin}\{\lambda x\, y.\neg\, p_q\}\ xs\ [\ y \mid y \leftarrow ys, p_r\ ], qs'\ ] \qquad \text{(ANTIJOIN-12)}$$

The primary introduction rules that we use can be summarized as follows:

- An introduction rule for `thetajoin{}` replaces consecutive generators in a qualifier list if a matching predicate is present.
- Rule ANTIJOIN-12 and similar introduction rules for the `antijoin{}` combinator rewrite universal quantification expressed with the boolean aggregate `and` as described by Claussen *et al.* [Cla+97].

- An analogous set of introduction rules for `semijoin{}` covers existential quantification.
- Introduction rules for `nestjoin{}` cover correlated subqueries in the head of a comprehension as well as consecutive generators and predicates. As demonstrated in Query Q9, this makes `nestjoin{}` a particularly versatile tool that covers a wide range of queries: queries constructing nested data structures from flat tables as well as complex predicates involving aggregates, for instance.

Introduction rules that we employ are mostly based on those described by Grust [Gru99] and Grust and Scholl [GS99]. We omit a detailed discussion of introduction rules as it offers no additional insight beyond their work. We list the complete set of rules in Appendix B.

The correctness of introduction rules can be established by replacing the join combinator in a rewritten term with its defining equation and using normalization rules to simplify the resulting term. We refer to Grust [Gru99] for a detailed account of this argument.

5.2.5 *Fusing Grouping and Aggregation*

In flat relational query languages, grouping and aggregation are necessarily entwined. Due to the flat data model, groups can not be represented explicitly and have to be aggregated. In nested query languages, though, grouping and aggregation are orthogonal concepts. Groups are produced *e.g.* by `nestjoin{}` (*binary* grouping) or `group` (*unary* grouping). They may then be consumed by `reduce{}` but may also appear in the query result or undergo transformations before aggregation. Query Q1 demonstrates some of these alternatives.

Query flattening readily lowers the combination of grouping and aggregation to flat queries: groups — produced by `nestjoin{}` or `group` — are simply nested lists represented by flat segment vectors. Applying `reduce{}` to those nested lists maps to data-parallel aggregation on segments, *i.e.* $\mathcal{SL}$ operators `reduceV{}` and `unboxV{}` (Section 4.3.3.1). In our compositional lowering scheme, this division is necessary to support grouping and aggregation as orthogonal concepts.

We assume that any relational query engine offers efficient primitives for grouped aggregation. If grouping and aggregation concur as in Query Q11, the compositional lowering scheme leads to non-idiomatic relational queries that are hard to map to those primitives.

```
[ ⟨g.1, sum [ x.v | x ← g.2 ], length g.2⟩ | g ← group xs ]      (Q11)
```

To obtain efficient backend code, we would have to recover backend primitives from non-idiomatic low-level code emitted for the combination of `groupV`, `reduceV{}` and `unboxV{}`.

Instead, we follow the same approach as for correlated iteration. We rewrite specific high-level patterns in $\mathcal{CL}$ queries into specialized combinators for grouped aggregation that lower directly to corresponding backend primitives. Specifically, we add rewrite rules for *fold-group fusion* [Ale+15]: we rewrite pairs of grouping combinators and `reduce{}` into `groupagg{}` and `groupjoin{}` combinators defined in Figures 44 and 45. These combinators are based on `group` and `nestjoin{}`, respectively, but extend the result with an aggregate of each group. Clearly, `groupagg{}` corresponds to usual relational grouping operators. `groupjoin{}` is a variant of *binary grouping*

$$
\begin{aligned}
&\texttt{groupjoin}\{s_p, s_z, s_f\}\ e_1\ e_2 = [\ \langle \texttt{x.1}, \texttt{reduce}\{s_z, s_f\}\ \texttt{x.2}\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid \texttt{x} \leftarrow \texttt{nestjoin}\{s_p\}\ e_1\ e_2\ ] \\
&\texttt{groupagg}\{s_z, s_f\}\ e = [\ \langle \texttt{x.1}, \texttt{x.2}, \texttt{reduce}\{s_z, s_f\}\ \texttt{x.2}\rangle \\
&\qquad\qquad\qquad\qquad \mid \texttt{x} \leftarrow \texttt{group}\ e\ ]
\end{aligned}
$$

Figure 44: Definition of $\mathcal{CL}$ grouping combinators, expressed in $\mathcal{CL}$.

$\mathcal{CL}$-TY-GROUPAGG

$$\frac{\Gamma \vdash e : [\langle \tau, \delta\rangle] \qquad \vdash s_z : \delta_a \qquad \vdash s_f : \delta_a \to \tau \to \delta_a}{\Gamma \vdash \texttt{groupagg}\{s_z, s_f\}\ e : [\langle \delta, [\tau], \delta_a\rangle]}$$

$\mathcal{CL}$-TY-GROUPJOIN

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : [\tau_1] \qquad \Gamma \vdash e_2 : [\tau_2] \\ \vdash s_p : \tau_1 \to \tau_2 \to \texttt{Bool} \qquad \vdash s_z : \delta_a \qquad \vdash s_f : \delta_a \to \langle \tau_1, \tau_2\rangle \to \delta_a\end{array}}{\Gamma \vdash \texttt{groupjoin}\{s_p, s_z, s_f\}\ e_1\ e_2 : [\langle \tau_1, \delta_a\rangle]}$$

Figure 45: Typing rules for $\mathcal{CL}$ grouping combinators.

operators that extend Steenhagen's nestjoin [SAB94] with a function (usually an aggregation function) that is applied to each group [Gru99; CM93; MHM04; MN11].

A number of straightforward introduction rules employ `groupagg{}` and `groupjoin{}` for aggregates encountered in the head of comprehensions as well as subsequent qualifiers. We list rules for fold-group fusion in Appendix B.

Note that the result of combinator `groupagg{}` includes the original groups next to the aggregate results. This accounts for queries in which groups are consumed by multiple aggregates (*e.g.* Query Q11). Rewrite rules gradually merge aggregates into the grouping combinator while preserving other references to the groups:

```
[ ⟨g.1, sum [ x.v | x ← g.2 ], length g.2⟩ | g ← group xs ]
≡   { GROUPAGG-HEAD }
  [ ⟨g.1, g.3, length g.2⟩ | g ← groupagg{0, λs e.s + e.a} xs ]
≡   { GROUPAGG-HEAD-EXTEND }
  [ ⟨⟨g.1, g.2, g.3.1⟩.1, ⟨g.1, g.2, g.3.1⟩.3, g.3.2⟩
  | g ← groupagg{⟨0, 0⟩, λs e.⟨s.1 + e.a, s.1 + 1⟩} xs ]
≡   { partially evaluate record selectors }
  [ ⟨g.1, g.3.1, g.3.2⟩
  | g ← groupagg{⟨0, 0⟩, λs e.⟨s.1 + e.a, s.1 + 1⟩} xs ]
```

Merging individual `reduce{}` occurrences one by one simplifies rewrite rules: we only consider one occurrence at a time and do not have to check whether groups are preserved explicitly in the query result. Admittedly, we pay for the simplicity of rules with a somewhat complex definition of `groupagg{}`. However, in Section 5.4, we show that this complexity is easily resolved during shredding.

5.2.6 *Predicate Pushdown*

A standard heuristic of logical query optimization is to apply predicates as early as possible. This heuristic is convenient to express during comprehension rewriting.

- In a sequence of comprehension qualifiers, a guard expression p can be freely moved to the left across other guards and generators. Obviously, we must not move guards over generators that bind variables occurring free in p.
- Guard expressions in comprehensions can be pushed into the argument of list combinators. For instance, the following rule pushes predicate p through the `semijoin{}` combinator if the only variable that occurs free in p is x:

$$\texttt{[h | qs, x} \leftarrow \texttt{semijoin\{s\} xs ys, p x, qs']}$$
$$\rightsquigarrow$$
$$\texttt{[h | qs, x} \leftarrow \texttt{semijoin\{s\} [x | x} \leftarrow \texttt{xs, p x] ys, qs']}$$

(PUSH-SEMIJOIN)

Similar rewrite rules push predicates into the input of other list combinators.

5.2.7 *Running Example*

We put join combinators to use to eliminate correlated nested iteration in our runnning example. We start with the fully normalized query Query Q1 and rewrite it in a completely mechanical sequence of transformations. Rewrite steps are shown in detail in Figure 46.

(1) We introduce a `nestjoin{}` combinator to unnest the correlated subquery in the predicate. For each order record, `nestjoin{}` produces the matching lineitem records required for the predicate.

(2) To avoid materializing those groups of matching lineitems, we fuse the `sum` aggregate into the `nestjoin{}` combinator. Note that the predicate is reduced to a scalar comparison on the `groupjoin{}` result.

(3) Next, we introduce another `nestjoin{}` combinator to unnest the correlated subquery in the head of the comprehension.

(4) So far, the `nestjoin{}` stacked on top of the `groupjoin{}` considers *all* orders since the `groupjoin{}` merely adds the aggregate result to each order. We push the predicate `5<w.3` into the left input of `nestjoin{}` to eliminate those orders that do not pass the predicate.

After optimization, all correlated nested iteration in Query Q12 is encapsulated in join combinators. A sequence of join combinators filters base tables and constructs a (nested) intermediate result that is consumed by the head of the outer comprehension. The nested comprehension in the head merely iterates over a nested list constructed by `nestjoin{}`.

The optimized query contains none of the problematic properties discussed in the beginning of this chapter. It does not contain nested base table references that would lead to data replication. Also, it does not feature variable references across iteration scopes that introduce environment lifting. Those are eliminated by the `nestjoin{}` combinator (Section 5.2.4.1).

```
[ ⟨o, sort [ ⟨l, l.sd - o.od⟩ | l ← ls, l.ok = o.ok ]⟩
| o ← os
, 5 < sum [ l.sd - o.od | l ← ls, l.ok = o.ok ] ]
≡   { NESTJOIN-GUARD (1) }
  [ ⟨u.1, sort [ ⟨l, l.sd - u.1.od⟩ | l ← ls, l.ok = u.1.ok ]⟩
  | u ← nestjoin{λo l.l.ok = o.ok} os ls
  , 5 < sum [ v.2.sd - v.1.od | v ← u.2 ] ]
≡   { GROUPJOIN-GUARD (2) }
  [ ⟨w.1, sort [ ⟨l, l.sd - w.1.od⟩ | l ← ls, l.ok = w.1.ok ]⟩
  | w ← groupjoin{ λo l.l.ok = o.ok, 0,
                   λa b.(a.2.sd - a.1.od) + b} os ls
  , 5 < w.3 ]
≡   { NESTJOIN-HEAD (3) }
  [ ⟨s.1.1, sort [ ⟨t.2, t.2.sd - t.1.1.od⟩ | t ← s.2 ]⟩
  | s ← nestjoin{λw l.l.ok = w.1.ok}
           (groupjoin{ λo l.l.ok = o.ok, 0,
                       λa b.(a.2.sd - a.1.od) + b} os ls)
           ls
  , 5 < s.1.3 ]
≡   { push down predicate (4) }
  [ ⟨s.1.1, sort [ ⟨t.2, t.2.sd - t.1.1.od⟩ | t ← s.2 ]⟩
  | s ← nestjoin{λw l.l.ok = w.1.ok}
           [ r
           | r ← groupjoin{ λo l.l.ok = o.ok, 0,                  (Q12)
                            λa b.(a.2.sd - a.1.od) + b} os ls
           , 5 < r.3 ]
           ls
  ]
```

Figure 46: Introducing join and grouping combinators in Query Q1.

5.2.8 *Rewrite Strategy*

This chapter focuses on providing *tools* for logical query optimization and integrating them with query flattening. Devising a holistic optimization strategy based on those tools is beyond the scope of this work. In principle, we provide all tools (*e.g.* join combinators, normalization, lifting loop-invariant expressions) to leverage the complete body of work on logical query optimization for orthogonal query languages.

In the rewriting of our running example in Figure 46, we have discussed a fixed sequence of rule applications. In general, though, for one expression multiple rewrite steps can apply. For example, in step (1), we may unnest the comprehension head first instead of starting with the guard. In general, ambiguities may be resolved based on a cost model. In the scope of this work, however, we follow a simple approach: we resolve those ambiguities from the start by giving priority to rule groups over others. Priorities are assigned based on a set of simple heuristics.

First, we assume that normalization is always beneficial and subsequent optimizations should be applied to fully normalized queries. Normalization reshapes comprehensions into the canonical forms expected by introduction rules. Furthermore, normalization eliminates redundant list nesting. In the following example, we find an immediate match for Rule NESTJOIN-HEAD.

```
concat [ [ ⟨x, y⟩ | y ← ys, x.a = y.b ] | x ← xs ]
≡   { NESTJOIN-HEAD }
  concat [ [ y | y ← x.2 ] | x ← nestjoin{λx y.x.a = y.b} xs ys ]
≡   { NORM-ID }
  concat [ x.2 | x ← nestjoin{λx y.x.a = y.b} xs ys ]
```

The resulting query enables a more efficient evaluation of the nested correlated iteration. Employing the `nestjoin{}` combinator avoids data replication and environment lifting. However, creating nested lists which are subsequently flattened with `concat` is clearly redundant. In this example, we are better off normalizing the query first and then employing a flat join combinator.

```
concat [ [ ⟨x, y⟩ | y ← ys, x.a = y.b ] | x ← xs ]
≡   { NORM-CONCAT }
  [ ⟨x, y⟩ | x ← xs, y ← ys, x.a = y.b ]
≡   { THETAJOIN }
  [ xy | xy ← thetajoin{λx y.x.a = y.b} xs ys ]
≡   { NORM-ID }
  thetajoin{λx y.x.a = y.b} xs ys
```

Hence, we give priority to comprehension normalization and partial evaluation over introduction rules for join combinators.

Furthermore, we assume that flat join operators (*e.g.* `antijoin{}`) should be preferred over nested joins. In the following example, `nestjoin{}` covers

the correlation part of the universal quantification. Subsequently, we employ `groupjoin{}` to fuse the boolean aggregate with the join.

```
[ x | x ← xs, and [ y.b < 42 | y ← ys, x.a = y.b ] ]
≡  { NESTJOIN-GUARD }
  [ x.1 | x ← nestjoin{λx y.x.a = y.b} xs ys
       , and [ y.2.b < 42 | xy ← x.2 ] ]
≡  { GROUPJOIN-GUARD }
  [ x.1 | x ← groupjoin{ λx y.x.a = y.b,
                         True, λa y.a ∧ (y.2.b < 42)} xs ys
       , x.2 ]
```

While the rewritten query is clearly an improvement, universal quantification is expressed in a very general way. First, `groupjoin{}` returns *all* elements of `xs` with the associated boolean aggregate. Only subsequently are those elements filtered based on the aggregate. Alternatively, we may rewrite into the specialized `antijoin{}` that fuses those two steps.

```
[ x | x ← xs, and [ y.b < 42 | y ← ys, x.a = y.b ] ]
≡  { ANTIJOIN-15 }
  [ x | x ← antijoin{λx y.x.a = y.b} xs [ y | ys, ¬ (y.b < 42) ] ]
≡  { NORM-ID }
  antijoin{λx y.x.a = y.b} xs [ y | ys, ¬ (y.b < 42) ]
```

We assume that the latter alternative based on `antijoin{}` performs better [Cla+97]. Rewriting into `groupjoin{}` only serves as a fallback for quantifiers that are not eligible for `antijoin{}`.

The implementation of query flattening that forms the basis for the experimental evaluation in Chapter 8 orders rewrites according to those considerations and standard heuristics. We structure rewrite rules into the following sequence.

1. Partial evaluation and comprehension normalization are applied exhaustively.
2. Predicates are pushed into the argument of list combinators.
3. Next, we detect flat joins. Here, we give priority to `semijoin{}` and `antijoin{}` over `thetajoin{}` matches.
4. Grouping and aggregation are fused.
5. Loop-invariant expressions are extracted from the head of comprehensions. The resulting `let`-bindings are moved up as far as possible.
6. Nested correlated comprehensions are unnested with `nestjoin{}`. We first search for matches in the qualifiers before searching in the head of the comprehension.

If a rewrite rule matches in one of these steps, we restart the sequence. We apply this sequence iteratively until we reach a fixed point. Rewriting typically assembles a tree of join and grouping operators in the generator of a comprehension. The resulting queries resemble the classical pattern of join graphs with plan tails [GMR10].

5.2.9 *Related Work on Optimization*

So far, none of the optimization techniques described in this chapter are original. As stated initially, we adapt well-known work on logical query optimization for nested or complex-object query languages.

Normalization rules on comprehensions are ubiquitous in the literature on orthogonal, comprehension-based collection programming and query languages. [Wad90; Gia+13; Ale+15; FM00; CLW14a; Gru99; GS99; Won96; Won94]. Both Grust [Gru99; GS99] and Fegaras [FM00] include an additional normalization rule that unnests existential quantifiers in an idempotent monad or monoid (*e.g.* the *set* monad or monoid). This rule is not applicable in our list-based language. Instead, we introduce list-based semi-join operators for existential quantifiers (Section 5.2.4).

Factoring loop-invariant expressions out of iterators is commonly employed to unnest uncorrelated subqueries [CM93; Ste95]. In a relational setting, it covers constant *type-A* queries according to Kim's classification of nested queries [Kim82]. In the context of the flattening transformation, Palmer and Prins [PP95] as well as Keller and Simons [KS96] include factoring as part of flattening: specialized flattening rules move constant expressions out of iterators. We do not inflate lifting with additional rules but employ a simple source-to-source translation prior to flattening that has the same effect.

Grust [Gru99] as well as Grust and Scholl [GS99] explore hybrid rewriting of comprehensions and combinators. Join combinators are introduced by rewrite rules for specific patterns of comprehension generators. They show that much prior work on optimization originally described for nested algebras can be understood and rephrased in terms of the (monad) comprehension calculus.

5.3 LIFTING JOIN COMBINATORS

Some optimizations (*e.g.* normalization, hoisting loop-invariant expressions) do not extend the language $\mathcal{CL}$ and are thus readily supported by *Query Flattening* as described in Chapter 4. However, our approach crucially relies on $\mathcal{CL}$ join and grouping combinators. Combinator-based optimization of $\mathcal{CL}$ queries is only feasible if we can integrate those combinators into query flattening and lower them to efficient flat queries.

As a first step towards integrating join and grouping combinators into query flattening, we describe the lowering to $\mathcal{FL}$. In the rewriting of our running example Query Q1, join combinators are located in the generator of the top-level comprehension and evaluated only once. In general, though, we can not rely on having combinators appear only at the top-level. Introduction rules for combinators reflect the orthogonal nature of $\mathcal{CL}$ and may match deeply nested comprehensions. As a consequence, we encounter join combinators that occur in the head of a comprehension and are applied to each element of a list.

We demonstrate this with the following example that features a universally quantified comprehension in the head of an outer comprehension. Note that the inner comprehension [p_2 y z | z ← zs, p_3 x z] depends on

both variables x and y, *i.e.* both enclosing comprehensions. Two rewrite steps replace the quantifier predicate with the antijoin{} combinator:

```
[ [ y | y ← ys, p1 x y, and [ p2 y z | z ← zs, p3 x z ] ] | x ← xs ]
≡  { NESTJOIN-HEAD }
   [ [ y.2 | y ← xy.2, and [ p2 y.2 z | z ← zs, p3 y.1 z ] ]
   | xy ← nestjoin{p1} xs ys ]
≡  { ANTIJOIN-16 }
   [ [ y.2 | y ← antijoin{λy z.¬(p2 y.2 z) ∧ p3 y.1 z} xy.2 zs ]
   | xy ← nestjoin{p1} xs ys ]
```

(Q13)

The first rewrite utilizes the nestjoin{} combinator to encapsulate the dependency between x and y: the individual groups xy.2 pair a binding for x with all corresponding bindings for y. By eliminating the dependency on x in the inner comprehension, the quantifier predicate is decoupled from the outer generator. This enables the second rewrite to introduce an antijoin{} combinator nested in the head of the outer comprehension that applies to each element of the result of nestjoin{}.

Such iterated join and grouping combinators naturally occur when rewriting complex, nested comprehensions. Furthermore, recall from Section 4.1 that comprehensions are desugared to $\mathcal{CL}_{\mathrm{d}}$ such that combinators occur *exclusively* in the head of iterators. Hence, dealing with the iterated application of join and grouping combinators is paramount.

Fortunately, we do not have to introduce any new concepts here. In Section 4.2, we describe how to deal with arbitrary combinators that are applied to all elements of a list in parallel: $\mathcal{L}[\![-]\!]$ maps combinator applications nested in iterators to applications of *lifted* data-parallel combinators $\mathrm{p}^{\uparrow}$ (Rule LIFT-BUILTIN). With scalar parameters fixed, join and grouping combinators appear as regular built-in list combinators. The $\mathcal{L}[\![-]\!]$ transformation lowers them via Rule LIFT-BUILTIN — no different from append, for example. After rewriting, Query Q13 lowers to $\mathcal{FL}$ as follows[1]:

```
[ [ y.2 | y ← antijoin{_} xy.2 zs ]
| xy ← nestjoin{_} xs ys ]
≡  { DESUGAR-TOP }
   concat [ [ [ y.2 | y ← antijoin{_} xy.2 zs ]
              | xy ← nestjoin{_} xs ys ]
            | z ← [⟨⟩] ]
≡  { L[[−]] }
   concat ( let z = [⟨⟩]
            in let xy = ⟦ nestjoin{_} (xs ⋉1 z) (ys ⋉1 z) ⟧1
               in let y = ⟦ antijoin{_} ⟦ xy.2 ⟧2 (zs ⋉2 xy) ⟧2
                  in ⟦ y.2 ⟧3 )
```

The application of nestjoin{} at iteration depth 1 maps to the lifted combinator $\texttt{nestjoin\{\}}^{\uparrow}$. Likewise, the antijoin{} combinator at iteration depth 2 maps to $\texttt{antijoin\{\}}^{\uparrow}$. Integrating additional combinators into query flattening comes down to defining those corresponding lifted combinators.

1 We omit join predicates (replaced with _) to increase readability. Join predicates are not affected by lifting.

The concept of lifted or data-parallel combinators easily extends to joins and grouping. Like all lifted combinators, lifted join and grouping combinators apply to lists of arguments. Join combinators apply to two lists of lists and join corresponding elements of the outer lists to produce a list of results. Combinator `antijoin{}`, for example, behaves as follows:

```
  antijoin{λx y.x > y}↑ [[4, 7, 8], [20, 30], [500, 600]]
                        [[5, 6], [], [100, 200, 700]]
≡ [[4], [20, 30], []]
```

An alert reader may notice that Query Q13 features the iterated list constant `zs`. Indeed, lifting replicates this list constant for each group produced by `nestjoin{}` to provide for a uniform data-parallel evaluation. This is an instance of data replication that we can not avoid with the tools introduced in this chapter. However, we discuss a solution in Chapter 7.

5.3.1 *Indexed Semantics*

In Section 4.2.4 we have defined the semantics of $\mathcal{FL}$ and its lifted combinators based on indexed lists. Against this background, the semantics of our new lifted join and grouping combinators are straightforward to define. For `groupagg{}`$^{\uparrow}$, we simply extend the interpretation of `group{}`$^{\uparrow}$ to fold the groups:

$$\begin{aligned}
&\mathcal{F}[\![\texttt{groupagg\{}s_z\texttt{, }s_f\texttt{\}}^{\uparrow}\ e]\!]_\rho = \\
&\quad [\ \langle \mathsf{k} = \mathsf{xs.k}, \mathsf{p} = [\ \langle \mathsf{k} = \mathsf{g.1}, \mathsf{p} = \langle \mathsf{g.1}, \mathsf{foldl}\ ([\![s_f]\!] \oslash \pi_\mathsf{p})\ [\![s_z]\!]\ \mathsf{xs.p} \\
&\qquad\qquad | \ \mathsf{g} \leftarrow \mathsf{groupWith}\ (\pi_1 \circ \pi_\mathsf{p})\ \mathsf{xs.p}\ ] \rangle \\
&\quad | \ \mathsf{xs} \leftarrow \mathcal{F}[\![e]\!]_\rho\ ]
\end{aligned} \tag{$\mathcal{FL}$-GROUPAGG}$$

Join combinators like `antijoin{}`$^{\uparrow}$ that consume two lists of arguments utilize indexes of the outer lists to identify corresponding arguments, *i.e.* inner lists. Those inner lists are joined based on the scalar predicate.

$$\begin{aligned}
&\mathcal{F}[\![\texttt{antijoin\{}s\texttt{\}}^{\uparrow}\ e_1\ e_2]\!]_\rho = \\
&\quad [\ \langle \mathsf{k} = \mathsf{xs.k}, \mathsf{p} = [\ \mathsf{x} \mid \mathsf{x} \leftarrow \mathsf{xs.p}, \mathsf{and}\ [\ \neg([\![s]\!]\ \mathsf{x.p}\ \mathsf{y.p}) \mid \mathsf{y} \leftarrow \mathsf{ys.p}\ ]\ ] \rangle \\
&\quad | \ \mathsf{xs} \leftarrow \mathcal{F}[\![e_1]\!]_\rho, \mathsf{ys} \leftarrow \mathcal{F}[\![e_2]\!]_\rho, \mathsf{xs.k} = \mathsf{ys.k}\ ]
\end{aligned} \tag{$\mathcal{FL}$-ANTIJOIN}$$

For `thetajoin{}`$^{\uparrow}$ and `nestjoin{}`$^{\uparrow}$, we additionally combine inner indexes of both arguments to derive unique indexes for the individual inner lists in the join result.

$$\begin{aligned}
&\mathcal{F}[\![\texttt{thetajoin\{}s\texttt{\}}^{\uparrow}\ e_1\ e_2]\!]_\rho = \\
&\quad [\ \langle \mathsf{k} = \mathsf{xs.k}, \mathsf{p} = [\ \langle \mathsf{k} = \langle \mathsf{x.k}, \mathsf{y.k} \rangle, \mathsf{p} = \langle \mathsf{x.p}, \mathsf{y.p} \rangle \rangle \\
&\qquad\qquad | \ \mathsf{x} \leftarrow \mathsf{xs.p}, \mathsf{y} \leftarrow \mathsf{ys.p}, [\![s]\!]\ \mathsf{x.p}\ \mathsf{y.p}\ ] \rangle \\
&\quad | \ \mathsf{xs} \leftarrow \mathcal{F}[\![e_1]\!]_\rho, \mathsf{ys} \leftarrow \mathcal{F}[\![e_2]\!]_\rho, \mathsf{xs.k} = \mathsf{ys.k}\ ]
\end{aligned} \tag{$\mathcal{FL}$-THETAJOIN}$$

$$
\begin{aligned}
&\mathcal{F}[\![\texttt{nestjoin\{s\}}^{\uparrow}\ e_1\ e_2]\!]_\rho = \\
&\quad [\ \langle \mathsf{k} = \mathsf{xs.k}, \mathsf{p} = [\ \langle \mathsf{x.k}, [\ \langle \mathsf{k} = \langle \mathsf{x.k}, \mathsf{y.k}\rangle, \mathsf{p} = \langle \mathsf{x.p}, \mathsf{y.p}\rangle\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad | \ \mathsf{y} \leftarrow \mathsf{ys.p}, [\![\mathsf{s}]\!]\ \mathsf{x.p\ y.p}\ ]\rangle \qquad\qquad (\mathcal{FL}\text{-NESTJOIN}) \\
&\qquad\qquad\qquad\quad | \ \mathsf{x} \leftarrow \mathsf{xs.p}\ ]\rangle \\
&\quad | \ \mathsf{xs} \leftarrow \mathcal{F}[\![e_1]\!]_\rho, \mathsf{ys} \leftarrow \mathcal{F}[\![e_2]\!]_\rho, \mathsf{xs.k} = \mathsf{ys.k}\ ]
\end{aligned}
$$

We omit the definitions for `groupjoin{}`$^{\uparrow}$ and `semijoin{}`$^{\uparrow}$ which are straightforward variations of other definitions listed above.

5.4 SHREDDING JOIN COMBINATORS

Shredding (Section 4.3) lowers lifted $\mathcal{FL}$ combinators on nested lists to $\mathcal{SL}$ operators on flat segment vectors. In this section, we integrate join and grouping combinators into this lowering step. Join and grouping combinators do not extend the expressiveness of $\mathcal{CL}$ but merely fuse specific combinations of $\mathcal{CL}$ combinators. Hence, we do not have to invent anything new here: the shredding framework of Section 4.3 readily supports lowering these new combinators as well. We define $\mathcal{SL}$ combinators that provide the nested iteration core of those combinators. Shredding rules map $\mathcal{FL}$ join combinators to those and maintain the structure of segment vectors.

In Section 5.4.1 we sketch the shredding of join combinators and introduce the required $\mathcal{SL}$ operators. We then define the shredding rules precisely in Section 5.4.2. To conclude our discussion of optimizations in this chapter, we discuss the shredding of our running example Query Q1 in its optimized form Query Q12 into a plan of flat $\mathcal{SL}$ operators (Section 5.4.3). In Section 5.4.4, we define the semantics of $\mathcal{SL}$ join operators in terms of $\mathcal{ML}$.

5.4.1 *Segment Join Operators*

We extend $\mathcal{SL}$ with a small number of operators, each of which implements the iterative core of a $\mathcal{CL}$ join combinator. Typing rules for these operators are listed in Figure 47. In this section, we discuss the essential challenges in shredding $\mathcal{FL}$ join combinators: 1. the data-parallel application of joins to lists of operands, 2. the maintenance of list nesting in operands and 3. the shredding of combinators that introduce additional list nesting.

5.4.1.1 *Lifted Joins*

A lifted join combinator consumes two lists of arguments and joins each pair of arguments individually. Consider the following application of lifted combinator `antijoinV{}`$^{\uparrow}$ to arguments of type `[[Int]]`:

```
antijoin{λx y.x > y}↑ [[4,7,8],[20,30],[500,600]]
                      [[5,6],[],[100,200,700]]
≡ [[4],[20,30],[]]
```

The argument expressions are shredded into packages $\texttt{[[Int]}^{V_1}\texttt{]}^{V_0}$ and $\texttt{[[Int]}^{V_2}\texttt{]}^{V_0'}$. Each inner list maps to a segment of the inner vectors V_1 or V_2. A $\mathcal{SL}$ implementation of `antijoinV{}`$^{\uparrow}$ joins corresponding pairs of segments in these inner vectors.

$\mathcal{SL}$-TY-THETAJOIN

$$\frac{\Gamma \vdash V_1 : \mathrm{D}\,(\alpha, \beta_1, \gamma_1) \qquad \Gamma \vdash V_2 : \mathrm{D}\,(\alpha, \beta_2, \gamma_2) \qquad \vdash s : \gamma_1 \to \gamma_2 \to \texttt{Bool}}{\Gamma \vdash \texttt{thetajoinV\{s\}}\ V_1\ V_2 : (\mathrm{D}\,(\alpha, \langle \beta_1, \beta_2 \rangle, \langle \gamma_1, \gamma_2 \rangle), \mathrm{R}\,\beta_1\,\langle \beta_1, \beta_2 \rangle, \mathrm{R}\,\beta_2\,\langle \beta_1, \beta_2 \rangle)}$$

$\mathcal{SL}$-TY-SEMIJOIN

$$\frac{\Gamma \vdash V_1 : \mathrm{D}\,(\alpha, \beta_1, \gamma_1) \qquad \Gamma \vdash V_2 : \mathrm{D}\,(\alpha, \beta_2, \gamma_2) \qquad \vdash s : \gamma_1 \to \gamma_2 \to \texttt{Bool}}{\Gamma \vdash \texttt{semijoinV\{s\}}\ V_1\ V_2 : (\mathrm{D}\,(\alpha, \beta_1, \gamma_1), \mathrm{F}\,\beta_1)}$$

$\mathcal{SL}$-TY-ANTIJOIN

$$\frac{\Gamma \vdash V_1 : \mathrm{D}\,(\alpha, \beta_1, \gamma_1) \qquad \Gamma \vdash V_2 : \mathrm{D}\,(\alpha, \beta_2, \gamma_2) \qquad \vdash s : \gamma_1 \to \gamma_2 \to \texttt{Bool}}{\Gamma \vdash \texttt{antijoinV\{s\}}\ V_1\ V_2 : (\mathrm{D}\,(\alpha, \beta_1, \gamma_1), \mathrm{F}\,\beta_1)}$$

$\mathcal{SL}$-TY-NESTJOIN

$$\frac{\Gamma \vdash V_1 : \mathrm{D}\,(\alpha, \beta_1, \gamma_1) \qquad \Gamma \vdash V_2 : \mathrm{D}\,(\alpha, \beta_2, \gamma_2) \qquad \vdash s : \gamma_1 \to \gamma_2 \to \texttt{Bool}}{\Gamma \vdash \texttt{nestjoinV\{s\}}\ V_1\ V_2 : (\mathrm{D}\,(\beta_1, \langle \beta_1, \beta_2 \rangle, \langle \gamma_1, \gamma_2 \rangle), \mathrm{R}\,\beta_1\,\langle \beta_1, \beta_2 \rangle, \mathrm{R}\,\beta_2\,\langle \beta_1, \beta_2 \rangle)}$$

$\mathcal{SL}$-TY-GROUPAGG

$$\frac{\Gamma \vdash V : \mathrm{D}\,(\alpha, \beta, \gamma) \qquad \vdash s_g : \gamma \to \gamma_g \qquad \vdash s_z : \gamma_a \qquad \vdash s_f : \gamma_a \to \gamma \to \gamma_a}{\Gamma \vdash \texttt{groupaggV\{}s_g, s_z, s_f\texttt{\}}\ V : \mathrm{D}\,(\alpha, \langle \alpha, \gamma_g \rangle, \langle \gamma_g, \gamma_a \rangle)}$$

$\mathcal{SL}$-TY-GROUPJOIN

$$\frac{\Gamma \vdash V_1 : \mathrm{D}\,(\alpha, \beta_1, \gamma_1) \qquad \Gamma \vdash V_2 : \mathrm{D}\,(\alpha, \beta_2, \gamma_2) \qquad \vdash s_p : \gamma_1 \to \gamma_2 \to \texttt{Bool} \qquad \vdash s_z : \gamma_a \qquad \vdash s_f : \gamma_a \to \langle \gamma_1, \gamma_2 \rangle \to \gamma_a}{\Gamma \vdash \texttt{groupjoinV\{}s_p, s_z, s_f\texttt{\}}\ V_1\ V_2 : \mathrm{D}\,(\alpha, \beta_1, \langle \gamma_1, \gamma_a \rangle)}$$

Figure 47: Typing rules for join and grouping operators in $\mathcal{SL}$.

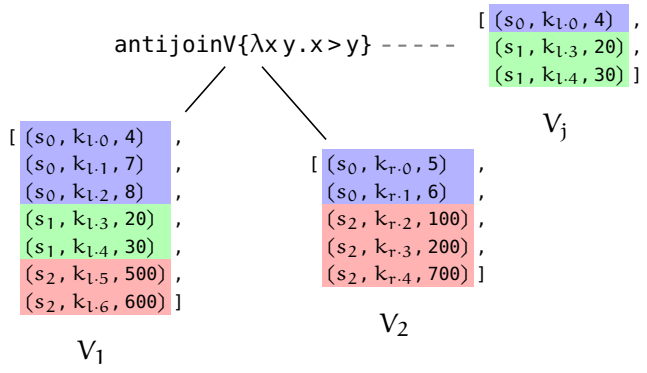


Figure 48: Implementing lifted join combinators on segment vectors: `antijoin{}`$^{\uparrow}$.

This situation is depicted in Figure 48. The $\mathcal{SL}$ `antijoinV{}` operator joins the inner vectors V_1 and V_2. Matching pairs of segments are joined individually. For example, in V_2 the segment identified by index s_1 is empty. The elements of this segment in V_1 do not find a join partner. Although elements of other segments in V_2 would match the universal quantification predicate, only elements of s_1 are considered. Hence, all elements of s_1 in V_1 appear in the result vector V_j.

Lowering join combinators to efficient relational plans is crucial to our optimization approach. Fortunately, the required per-segment join operators are easily derived from regular join operators. Recall that all arguments of any lifted combinator $p^{\uparrow}$ as well as its result share the same *list shape* – this property is guaranteed by the flattening transformation. The outer vectors representing those lists consequentially also have the same vector shape. Two properties follow from this:

1. Corresponding segments in the inner vectors are identified by having the same outer index. Matching corresponding segments of two inner vectors comes down to comparing scalar outer index values.

2. Evaluating lifted join combinators in $\mathcal{SL}$ exclusively affects the inner vectors — V_1 and V_2 in Figure 48. The segment structure of those inner vectors stays the same. The outer vector of any of the arguments accurately describes this segment structure and can serve as the outer vector of the result. A segment join operator is thus evaluated locally on two flat (inner) vectors and it's result does not have to be propagated to outer vectors[2].

These two properties hint towards the implementation of per-segment join operators like `antijoinV{}`. On individual pairs of segments, `antijoinV{}` behaves as an ordinary order-preserving antijoin operator. Based on such an operator, the segment join operator `antijoinV{}` can be easily implemented: it is sufficient to extend the join predicate with a comparison of outer indexes.

5.4.1.2 *Maintaining Nested Arguments*

Consider the following example in which one argument of `thetajoin{}`$^{\uparrow}$ features an additional layer of list nesting.

```
thetajoin{λx y.x=y.a}↑ [[1,2,4,1]]
                       [[⟨a=1,b=[10,20]⟩,
                         ⟨a=2,b=[30]⟩]]
≡ [[⟨1,⟨a=1,b=[10,20]⟩⟩,
   ⟨2,⟨a=2,b=[30]⟩⟩,
   ⟨1,⟨a=1,b=[10,20]⟩⟩]]
```

The $\mathcal{SL}$ implementation of this example is depicted in Figure 49. Operator `thetajoinV{}` generates the result vector V_j that contains the join result. Note that `thetajoinV{}` combines the inner indexes of V_1 and V_2 to derive unique inner indexes for V_j. Crucially, though, outer indexes are preserved.

The element of V_2 identified by $k_{r\cdot 0}$ appears in the join result two times. To keep the relation to the inner vector V_i consistent, the corresponding

2 This property only holds for an index-based vector model (Section 4.3.1). In a length-based representation, segment descriptors need to be modified if the size of segments in inner vectors change.

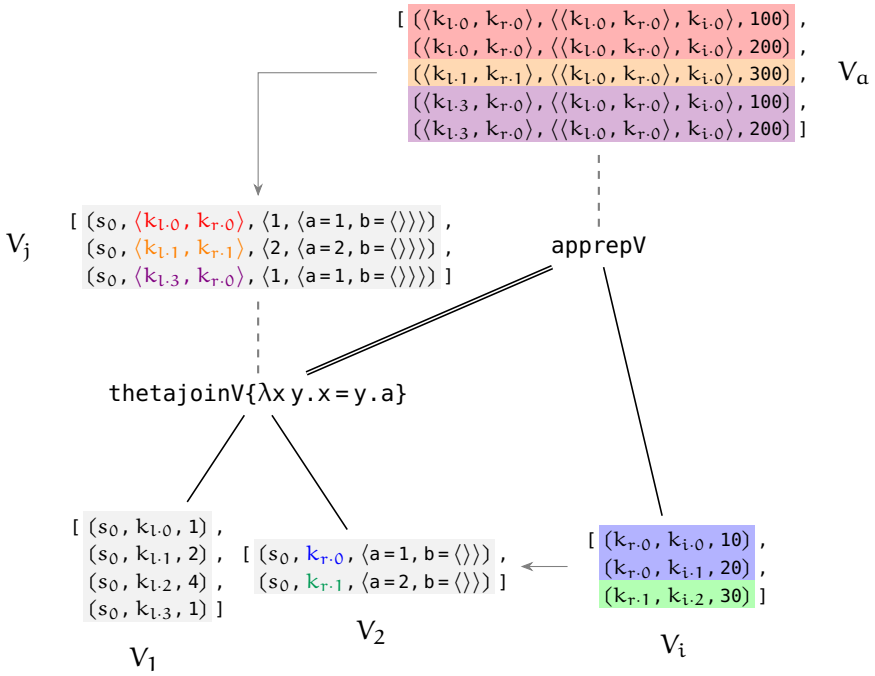


Figure 49: Maintenance of nested arguments for `thetajoin{}`$^{\uparrow}$.

segment in V_i needs to be replicated. Such changes to inner vectors are propagated just as described in Section 4.3.2: next to the actual result vector V_j, the join operator generates an *index transformation*:

$$[\ \langle \mathtt{f} = k_{r\cdot 0}, \mathtt{t} = \langle k_{l\cdot 0}, k_{r\cdot 0}\rangle\rangle\ ,\ \langle \mathtt{f} = k_{r\cdot 1}, \mathtt{t} = \langle k_{l\cdot 1}, k_{r\cdot 1}\rangle\rangle\ ,\ \langle \mathtt{f} = k_{r\cdot 0}, \mathtt{t} = \langle k_{l\cdot 3}, k_{r\cdot 0}\rangle\rangle\ ]$$

Applied to V_i by `apprepV`, this replication transformation aligns V_i with V_j. In the result V_a, segment $s_{r\cdot 0}$ is replicated and outer and inner indexes are adapted to match V_j.

5.4.1.3 *Combinators with Nested Results*

Combinators `nestjoin{}`$^{\uparrow}$ and `groupagg{}`$^{\uparrow}$ introduce additional list nesting. Fortunately, these operators are straightforward to implement on the segment vector model. They shred into simple bulk operators on flat vectors.

NESTED JOIN Consider an application `nestjoin{}`$^{\uparrow}$ e_1 e_2 of the lifted `nestjoin{}`$^{\uparrow}$ combinator. We assume that argument expressions e_1 and e_2 shred into the following packages:

$$e_1 \ \wr\ [[\mathtt{Int}]^{V_1}]^{V_o}$$

$$e_2 \ \wr\ [[\langle \mathtt{a{:}Int}, \mathtt{b{:}Text}\rangle]^{V_2}]^{V'_o}$$

Vectors V_1 and V_2 represent the elements of the inner lists. We know that the outer list structure of e_1 and e_2 is the same and that the corresponding outer vectors V_o and V'_o have the same vector shape.

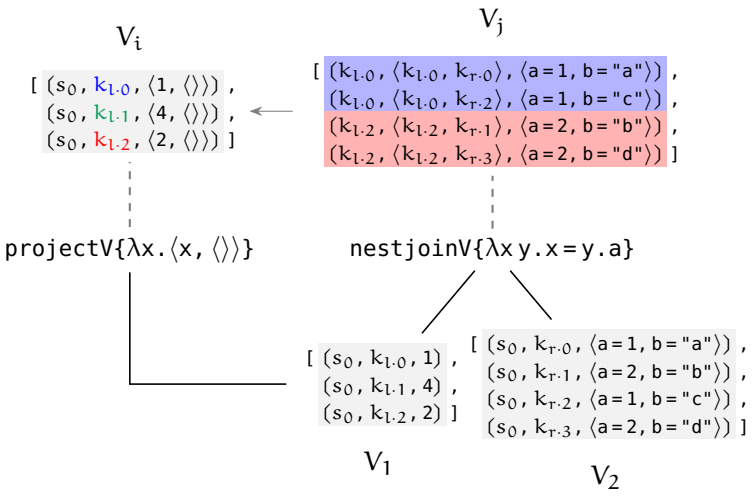


Figure 50: Implementing `nestjoin{}`$^{\uparrow}$ on segment vectors.

`nestjoin{}` computes the (possibly empty) list of matches in the right operand for each element of its left operand. The nested join results in the following package that references three vectors:

$$\texttt{nestjoin\{s\}}^{\uparrow}\ e_1\ e_2\ \texttt{\$}\ [[\langle \texttt{Int}, [\langle \texttt{Int}, \langle \texttt{a:Int,b:Text} \rangle\rangle]^{V_j} \rangle]^{V_i}]^{V_o''}$$

As the outermost list structure of the result does not change, V_o'' has the same vector shape as V_o and V_o', the outer vectors of the arguments[3]. Structure and content of the list of groups are encoded by vectors V_i and V_j, respectively.

Let the predicate be `λx y.x=y.a` and let arguments e_1 and e_2 evaluate to the following values:

$$e_1 \equiv \texttt{[[1, 4, 2]]}$$
$$e_2 \equiv \texttt{[[}\langle \texttt{a=1, b="a"} \rangle, \langle \texttt{a=2, b="b"} \rangle, \langle \texttt{a=1, b="c"} \rangle,$$
$$\langle \texttt{a=2, b="d"} \rangle \texttt{]]}$$

Given these arguments, computation of the inner vectors V_i and V_j is depicted in Figure 50. Vector V_i encodes the structure of groups while vector V_j encodes the content of all groups. Providing the *structure* in V_i is actually easy: as each element of the left nestjoin input maps to a group in the nestjoin result, vector V_1 readily describes the resulting structure. We merely have to insert the $\langle\rangle$ placeholder in the payload of V_1. To assemble the *content* of all groups in V_j, the $\mathcal{SL}$ operator `nestjoinV{}` joins matching segments of V_1 and V_2. All matches for one particular element of V_1 are assembled in a segment identified by that elements' inner index. This establishes the index relationship between V_i and V_j and thus the representation of the nested join result.

Note that despite the nested result of `nestjoin{}`, the corresponding vector operator `nestjoinV{}` is a regular flat join of two flat collections. The only difference to `thetajoinV{}` lies in the handling of outer indexes. We can safely assume that the implementation of `nestjoinV{}` poses no more challenge than that of `thetajoin{}`. If we are able to lower order-aware

3 Since the payload of these outer vectors is just a placeholder $\langle\rangle$, they do not only have the same vector shape but are actually identical.

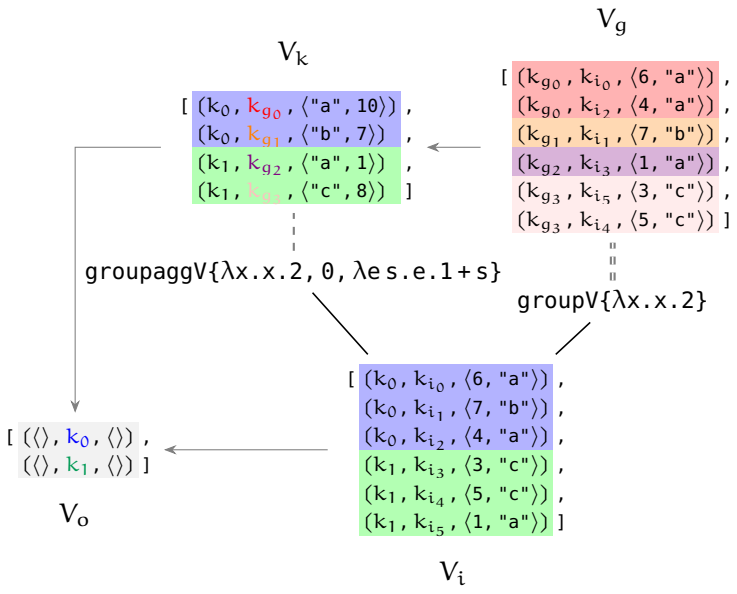


Figure 51: $\mathcal{SL}$ implementation of `groupagg{}`$^{\uparrow}$ (based on Figure 34).

segment joins to efficient relational algebra plans, we can support nested joins efficiently in query flattening. This enables the use of the `nestjoin{}` combinator as an essential tool in the optimization of nested queries.

GROUPED AGGREGATION Next to join combinators, we introduced $\mathcal{CL}$ combinators `groupagg{}` and `groupjoin{}` that fuse grouping and aggregation. To ease comprehension rewriting, they return both the groups as well as an aggregate of that group. During shredding, however, that complexity disappears: both combinators map to simple $\mathcal{SL}$ operators on flat vectors.

Consider the following application of `groupagg{}`$^{\uparrow}$ in which argument e has the package $[[\langle \texttt{Int}, \texttt{Text} \rangle]^{V_i}]^{V_o}$:

$$\texttt{groupagg\{0, } \lambda \texttt{e s.e.1 + s\}}^{\uparrow}\ e$$

Elements are grouped by the second pair component of type `Text`. According to the type of combinator `groupagg{}`$^{\uparrow}$ (see Figure 45), this application shreds into the following package:

$$[[\langle \texttt{Text}, [\texttt{Int}]^{V_g}, \texttt{Int} \rangle]^{V_k}]^{V_o}$$

Based on the example of Figure 34, we illustrate in Figure 51 how to obtain the result vectors V_k and V_g that represent the structure and content of groups, respectively. Vector operator `groupaggV{}` groups each segment of V_i independently and includes the aggregate for each group. Vector V_k contains an element for each group that features the grouping key as well as the aggregate value. Note that V_k and the outer vector returned by the `groupV{}` operator in Figure 34 have the same vector shape. Indeed, the structure of the groups computed in both examples is the same. Due to this correspondence, we can employ operator `groupV{}` to provide the group content in the inner vector V_g (identical to V_g in Figure 34). The outer vector also computed by `groupV{}` is not referenced. We have omitted the `projectV{}` operator that removes the `Text` grouping keys from V_g as well as the `projectV{}` operator on V_k that inserts $\langle\rangle$ into the payload as the placeholder for the nested list.

As demonstrated here, the effort to support the combinator `group{}` during shredding is minimal: we only extend $\mathcal{SL}$ with a simple flat, segment-based operator for grouped aggregation.

5.4.2 *Shredding Rules for Join Combinators*

In this section, we extend shredding (Section 4.3.3) with rules that handle join and grouping combinators.

Shredding rules for the $\mathcal{FL}$ combinators `thetajoin{}`$^{\uparrow}$, `semijoin{}`$^{\uparrow}$ and `antijoin{}`$^{\uparrow}$ essentially emit the corresponding $\mathcal{SL}$ join operator. As described in Section 5.4.1.1, for instance, lifted combinator `antijoin{}`$^{\uparrow}$ directly maps to operator `antijoinV{}` that implements the essential iteration pattern encapsulated in the combinator. Additionally, index transformation vectors obtained from those join operators are recursively applied to the element packages to maintain the structure of vectors that represent nested lists. For `thetajoin{}`$^{\uparrow}$, we employ $(\!|\ \_\ |\!)\ \_$ to replicate and filter segments in inner vectors as described in Section 5.4.1.2.

SHRED-THETAJOIN-LIFT

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \wr [[\rho_1]^{V_{i\cdot1}}]^{V_{o\cdot1}} \\ \Gamma \vdash e_2 \wr [[\rho_2]^{V_{i\cdot2}}]^{V_{o\cdot2}} \qquad \left[(V_j, I_1, I_2) \leftarrow \texttt{thetajoinV\{s\}}\ V_{i\cdot1}\ V_{i\cdot2}\right]\end{array}}{\Gamma \vdash \texttt{thetajoin\{s\}}^{\uparrow}\ e_1\ e_2 \wr [[\langle (\!|\rho_1|\!)_{I_1}, (\!|\rho_2|\!)_{I_2}\rangle]^{V_j}]^{V_{o\cdot1}}}$$

SHRED-SEMIJOIN-LIFT

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \wr [[\rho_1]^{V_{i\cdot1}}]^{V_{o\cdot1}} \\ \Gamma \vdash e_2 \wr [[\rho_2]^{V_{i\cdot2}}]^{V_{o\cdot2}} \qquad \left[(V_j, I) \leftarrow \texttt{semijoinV\{s\}}\ V_{i\cdot1}\ V_{i\cdot2}\right]\end{array}}{\Gamma \vdash \texttt{semijoin\{s\}}^{\uparrow}\ e_1\ e_2 \wr [[(\!|\rho_1|\!)_I]^{V_j}]^{V_{o\cdot1}}}$$

SHRED-ANTIJOIN-LIFT

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \wr [[\rho_1]^{V_{i\cdot1}}]^{V_{o\cdot1}} \\ \Gamma \vdash e_2 \wr [[\rho_2]^{V_{i\cdot2}}]^{V_{o\cdot2}} \qquad \left[(V_j, I) \leftarrow \texttt{antijoinV\{s\}}\ V_{i\cdot1}\ V_{i\cdot2}\right]\end{array}}{\Gamma \vdash \texttt{antijoin\{s\}}^{\uparrow}\ e_1\ e_2 \wr [[(\!|\rho_1|\!)_I]^{V_j}]^{V_{o\cdot1}}}$$

Shredding rule SHRED-NESTJOIN-LIFT emits two simple operators (a flat join and a projection) to obtain the two vectors that make up the nested result of `nestjoin{}`$^{\uparrow}$.

SHRED-NESTJOIN-LIFT

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \wr [[\rho_1]^{V_{i\cdot1}}]^{V_{o\cdot1}} \\ \Gamma \vdash e_2 \wr [[\rho_2]^{V_{i\cdot2}}]^{V_{o\cdot2}} \qquad \begin{bmatrix}(V_j, I_1, I_2) \leftarrow \texttt{nestjoinV\{s\}}\ V_{i\cdot1}\ V_{i\cdot2} \\ V_m \qquad\qquad\quad \leftarrow \texttt{projectV\{}\lambda x.\langle x, \langle\rangle\rangle\texttt{\}}\ V_{i\cdot1}\end{bmatrix}\end{array}}{\Gamma \vdash \texttt{nestjoin\{s\}}^{\uparrow}\ e_1\ e_2 \wr [[\langle \rho_1, [\langle (\!|\rho_1|\!)_{I_1}, (\!|\rho_2|\!)_{I_2}\rangle]^{V_j}\rangle]^{V_m}]^{V_{o\cdot1}}}$$

Rule SHRED-GROUPJOIN-LIFT implements binary grouping on segment vectors.

SHRED-GROUPJOIN-LIFT

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \wr [[\rho_1]^{V_{i\cdot1}}]^{V_{o\cdot1}} \qquad \Gamma \vdash e_2 \wr [[\rho_2]^{V_{i\cdot2}}]^{V_{o\cdot2}} \\ \vdash z : \delta_a \qquad \left[V_a \leftarrow \texttt{groupjoinV\{}s_p, s_z, s_f\texttt{\}}\ V_{i\cdot1}\ V_{i\cdot2}\right]\end{array}}{\Gamma \vdash \texttt{groupjoin\{}s_p, s_z, s_f\texttt{\}}^{\uparrow}\ e_1\ e_2 \wr [[\langle \rho_1, \delta_a\rangle]^{V_a}]^{V_{o\cdot1}}}$$

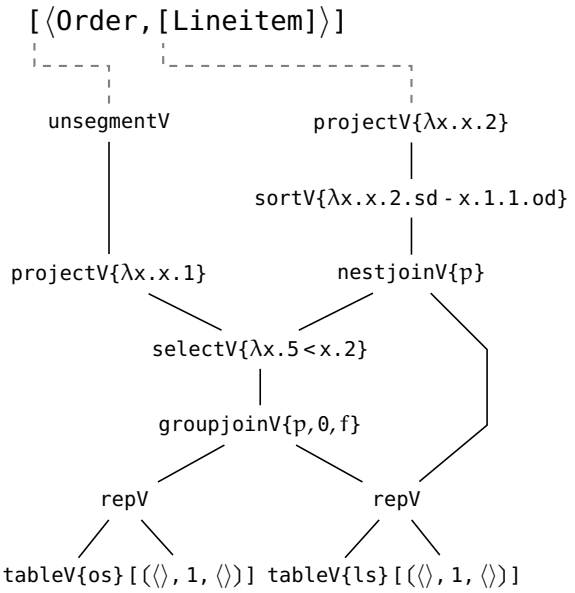


Figure 52: Flat $\mathcal{SL}$ plan for the running example Query Q12, obtained via shredding. Scalar functions for join predicate and aggregate abbreviated as p, and f.

As described earlier, rule SHRED-GROUPAGG-LIFT emits operators that compute both explicit groups and group aggregates. Note that we apply a sorting transformation to the group element type just as in SHRED-GROUP-LIFT.

SHRED-GROUPAGG-LIFT

$$\frac{\Gamma \vdash e \Rightarrow [[\langle \rho_1, \rho_2 \rangle]^{V_i}]^{V_o} \qquad \vdash z : \delta_a \qquad \begin{bmatrix} (\_, V_g, I) & \leftarrow \texttt{groupV\{}\lambda\texttt{x.x.2\}}\ V_i \\ V'_g & \leftarrow \texttt{projectV\{}\lambda\texttt{x.x.1\}}\ V_g \\ V_a & \leftarrow \texttt{groupaggV\{}\lambda\texttt{x.x.2},\ s_z,\ s_f\texttt{\}}\ V_i \\ V_k & \leftarrow \texttt{projectV\{}\lambda\texttt{x.}\langle\texttt{x.1},\langle\rangle,\texttt{x.2}\rangle\texttt{\}}\ V_a \end{bmatrix}}{\Gamma \vdash \texttt{groupagg\{}s_z,\ s_f\texttt{\}}^{\uparrow}\ e_1 \Rightarrow [[\langle \rho_2, [\langle \rho_1 \rangle_I]^{V'_g}, \delta_a \rangle]^{V_k}]^{V_o}}$$

The $\mathcal{CL}$ `groupagg{}` combinator enables step-by-step fusion of multiple aggregates as well as queries in which groups are both aggregated and retained. If groups are exclusively aggregated, however, the vector V_g produced by `groupV{}` is not required. In that case, variable V_k is not referenced and the statement binding it can be eliminated via dead-code analysis. No overhead is incurred.

5.4.3 *Shredding Running Example*

The combination of join combinators in $\mathcal{CL}$ queries and shredding produces reasonable flat $\mathcal{SL}$ plans. Figure 52 shows the $\mathcal{SL}$ plan for the optimized running example Query Q12. $\mathcal{SL}$ optimizations described in Section 4.3.4 have been applied to inline scalar expressions into `selectV{}` and `sortV{}` operators. Other than that, the plan is a direct output of the shredding translation.

5.4.4 *Semantics of Segment Join Operators*

We define list-based interpretations for segment join and grouping operators that integrate with the $\mathcal{SL}$ semantics described in Section 4.3.2.2. Next to

defining the semantics of our new operators precisely, the interpretations also hint towards the lowering of those operators to relational algebra.

Operator `thetajoinV{}` is defined based on a simple list comprehension over the input vectors V_1 and V_2 in ($\mathcal{SL}$-THETAJOIN). We extend predicate p with comparison of outer indexes to only consider elements of the same segment. We pair inner indexes of V_1 and V_2 to obtain unique inner indexes for the result. From V, we derive the index transformations I_1 and I_2 that map old indexes to new indexes. We have omitted the interpretation of `nestjoinV{}` which is essentially identical to ($\mathcal{SL}$-THETAJOIN) — the difference being that we use inner indexes of V_1 as outer indexes of the result.

$$
\begin{aligned}
&\mathcal{S}[\![\texttt{thetajoinV\{s\}}\ V_1\ V_2]\!]\rho = && (\mathcal{SL}\text{-THETAJOIN})\\
&\quad \mathsf{let}\ V = [\ (x.s, \langle x.k, y.k\rangle, \langle x.p, y.p\rangle)\\
&\qquad\qquad | \ x \leftarrow \rho(V_1), y \leftarrow \rho(V_2)\\
&\qquad\qquad , \ x.s = y.s, [\![s]\!]\ x.p\ y.p\ ]\\
&\qquad M_1 = [\ \langle f = x.k.1, t = x.k\rangle \mid x \leftarrow V\ ]\\
&\qquad M_2 = [\ \langle f = x.k.2, t = x.k\rangle \mid x \leftarrow V\ ]\\
&\quad \mathsf{in}\ (V, M_1, M_2)
\end{aligned}
$$

Likewise, ($\mathcal{SL}$-ANTIJOIN) interprets `antijoinV{}` with a simple list comprehension. We omit the interpretation of `semijoinV{}` which is a trivial variation that trades universal for existential quantification.

$$
\begin{aligned}
&\mathcal{S}[\![\texttt{antijoinV\{s\}}\ V_1\ V_2]\!]\rho = && (\mathcal{SL}\text{-ANTIJOIN})\\
&\quad \mathsf{let}\ \ V = [\ x \mid x \leftarrow \rho(V_1), \mathsf{and}\ [\ \neg\ ([\![s]\!]\ x.p\ y.p)\\
&\qquad\qquad\qquad\qquad\qquad\qquad | \ y \leftarrow \rho(V_2), x.s = y.s\ ]\ ]\\
&\qquad M = [\ x.k \mid x \leftarrow V\ ]\\
&\quad \mathsf{in}\ (V, M)
\end{aligned}
$$

Note that the core of ($\mathcal{SL}$-THETAJOIN) matches the calculus definition of the relational thetajoin operator $\bowtie$. Deriving the index transformations I_1 and I_2 matches a relational projection π on the join result. Likewise, ($\mathcal{SL}$-ANTIJOIN) directly matches the calculus definition of the relational operator $\triangleright$. This indicates that these segment join operators can be lowered rather directly to idiomatic fragments of relational algebra.

Operator `groupjoinV{}` is interpreted in ($\mathcal{SL}$-GROUPJOIN) with nested comprehensions. Groups are folded directly. As with the other segment join operators, the defining list comprehension resembles the calculus definition of a relational operator, namely the groupjoin operator as defined by Moerkotte and Neumann [MN11].

$$
\begin{aligned}
&\mathcal{S}[\![\texttt{groupjoinV\{}s_p, s_z, s_f\texttt{\}}\ V_1\ V_2]\!]\rho =\\
&\quad [\ \mathsf{let}\ a = \mathsf{foldl}\ ([\![s_f]\!]\ \oslash\ \pi_p)\ [\![s_z]\!]\ [\ \langle x.p, y.p\rangle\\
&\qquad\qquad\qquad\qquad\qquad | \ y \leftarrow \rho(V_2), x.s = y.s, [\![s_p]\!]\ x.p\ y.p\ ]\\
&\qquad \mathsf{in}\ (x.s, x.k, \langle x.p, a\rangle)\\
&\quad | \ x \leftarrow \rho(V_1)\ ]
\end{aligned}
$$

($\mathcal{SL}$-GROUPJOIN)

Finally, to implement `groupaggV{}`, ($\mathcal{SL}$-GROUPAGG) uses groupWith to group the individual segments and folds the resulting groups.

$$
\begin{aligned}
&\mathcal{S}[\![\texttt{groupaggV\{}s_g, s_z, s_f\texttt{\}}\ V]\!]\rho = \\
&\quad \mathsf{concat}\ [\ [\ (\mathsf{seg}.1\,,\ \langle \mathsf{seg}.1, \mathsf{g}.1\rangle\,,\ \mathsf{foldl}\ ([\![s_f]\!] \oslash \pi_p)\ [\![s_z]\!]\ \mathsf{seg}.2) \\
&\qquad\quad \mid \mathsf{g} \leftarrow \mathsf{groupWith}\ ([\![s_f]\!] \circ \pi_p)\ \mathsf{seg}.2\ ] \\
&\qquad \mid \mathsf{seg} \leftarrow \mathsf{segs}\ \rho(V)\ ]
\end{aligned}
$$

($\mathcal{SL}$-GROUPAGG)

5.4.4.1 *Segment Joins and Relational Joins*

At the beginning of this chapter, we start by encapsulating specific patterns of correlated nested iteration into join combinators. Our aim has been to map these patterns to efficient backend primitives and prevent the flattening transformation from enforcing a naive evaluation strategy. By peeling away abstractions from $\mathcal{CL}$ join combinators, we end up with $\mathcal{SL}$ segment join combinators. As we have seen in this section, the comprehension definition of all segment join operators closely resembles the calculus definitions of relational join operators. This correspondence indicates that $\mathcal{CL}$ join combinators indeed can be lowered almost directly to relational joins. This is a first verification of the optimization approach described in this chapter: we optimize queries in the high-level language $\mathcal{CL}$where the query structure is clearly visible and introduction of join combinators can be expressed conveniently. We then rely on query flattening to lower these combinators to a low-level flat query language.

We can reasonably expect that the simple iteration patterns of segment join operators are well supported by query engines. Recall, though, that we start with list-based $\mathcal{CL}$ joins and that segment operators are *order-preserving* operators defined on ordered vectors. The ordered semantics of join operators is the last remaining roadblock. We discuss the lowering of order-aware operators to unordered relational algebra in Chapter 6.

6 RELATIONAL BACKEND

Query Flattening (Chapter 4) eliminates the major obstacles that prevent an implementation of the query language $\mathcal{CL}$ on relational backends. Instead of nested iteration and nested data, we only face a flat representation of nested data and a number of rather simple flat, data-parallel operators. Optimizations described in Chapter 5 turn the nested-loop nature of flattened queries into iteration patterns (*e.g.* joins) closer to relational plans. Still, the segment vector model abstracts over a number of aspects that do not map to relational query processing in an obvious way.

- We require an efficient scheme for creating and maintaining indexes.
- Vectors are implicitly ordered and vector operators preserve the order of their inputs. The relational model, on the other hand, is restricted to unordered sets or multisets. We require a representation of ordered data in terms of unordered collections.
- Lifted list combinators (*e.g.* `sort`$^{\uparrow}$) are lowered to vector operators (*e.g.* `sortV{f}`) that observe the segmentation of a vector and perform the computation on each segment individually. These segment operators need to be lowered to regular relational bulk operators.

In this chapter, we take the final step towards relational queries: We describe an interpretation of segment vector operators in terms of relational algebra. Our goal is to devise idiomatic relational plans that are fit for execution without substantial rewriting efforts. We show that segment vector operators map to benign and idiomatic combinations of relational operators. We aim for a relational encoding of list order that does not burden the backend code with expensive order maintenance.

The remainder of this chapter is structured as follows: In Section 6.1 we describe the multiset algebra and define its static and dynamic semantics. In Section 6.2, we sketch the code generator: We discuss the implementation of segment vector operators with multiset operators as well as indexing schemes and the representation of order. Combining these aspects, we describe a relational code generator based on natural indexes and lazy order in Section 6.3. We conclude with a discussion of relational optimizations (Section 6.4) and SQL code generation (Section 6.5).

6.1 MULTISET ALGEBRA

We define a multiset algebra $\mathcal{MA}$ that serves as an intermediate language in the translation from vector operators to relational queries. $\mathcal{MA}$ is close to textbook relational algebra and reflects the capabilities of real-world query engines. Operators of the algebra are supported by any reasonably complete SQL:2003 database system. We find it convenient to deviate somewhat from standard notation and use lambda notation for operator arguments. For example, projection is expressed as π`{λx.s} q` where `q` is an algebra expression and `s` is an arbitrary scalar expression.

To enable a simple translation from vector operators, we use the same language for scalar expressions as for all other intermediate languages (Fig-

ure 8). All $\mathcal{MA}$ operators consume and produce multisets of type $\{\!|\delta|\!\}$ with scalar elements of type δ. Hence, multiset elements may be arbitrarily nested records. Nested records will prove helpful to formulate translation rules that are not burdened with tracking and renaming of relational columns. Nested records can be mapped to flat records in a subsequent translation (Section 6.5). In contrast to real-world relational systems, our data model does not include `NULL` values. Avoiding `NULL` simplifies the semantics of the algebra considerably. This restriction is not fundamentally necessary, though.

We define the typing rules for all algebra operators in Figure 53 and a list interpretation in Figure 54. The algebra features the core relational operators $\pi\{\}$, $\sigma\{\}$, $\cup$ and $\times$. Note that $\times$ produces *pairs* of elements from its operands and thus nested records. Explicit duplicate elimination on multisets can be expressed with operator δ.

The join operator $\ltimes\{\}$ combines $\times$ with an arbitrary scalar predicate. In $q_1 \ltimes\{\lambda x\,y.s\}\, q_2$, predicate expression s is evaluated with `x` and `y` bound to elements of operands q_1 and q_2, respectively. Semi- and antijoin operators $\ltimes\{\}$ and $\triangleright\{\}$ express existential and universal quantification. We also include the left outerjoin operator $⟕\{\}$. We follow Moerkotte and Neumann [MN11] and let $⟕\{\}$ attach an arbitrary scalar value instead of `NULL` for missing join partners. In $⟕\{s_p, s_z, s_r\}$, scalar function s_p is the join predicate while value s_z defines the value to be assigned for missing join partners. Scalar function s_r is applied to elements of the right operand and allows to restrict those values. Consider the following example.

$$\begin{aligned}&\{\!|\,\texttt{1},\texttt{2}\,|\!\}\ ⟕\{\lambda x\,y.=x\,y.\texttt{1},\texttt{42},\lambda x.x.\texttt{2}\}\ \{\!|\,\langle\texttt{1},\texttt{5}\rangle,\langle\texttt{3},\texttt{23}\rangle\,|\!\}\\ =\ &\{\!|\,\langle\texttt{1},\texttt{5}\rangle,\langle\texttt{2},\texttt{42}\rangle\,|\!\}\end{aligned}$$

With this definition of $⟕\{\}$, we can employ outer joins without having to consider `NULL` in the algebra. When translating $\mathcal{MA}$ to SQL:2003, $⟕\{\}$ can be simulated with `NULL`.

Operator $\Gamma\{\}$ expresses grouped aggregation. Given $\Gamma\{s_g, s_z, s_f\}$, scalar function s_g maps each element of the input multiset to a grouping key. Arguments s_z and s_f define folding of each group into a single scalar value. As an example, the following expression groups a collection of pairs by their first component and then computes the sum of their second pair components as well as the number of each groups' elements:

$$\begin{aligned}&\Gamma\{\lambda x.x.\texttt{1},\langle\texttt{0},\texttt{0}\rangle,\lambda x\,y.\langle\texttt{s}=+\,x.\texttt{2}\,y,\texttt{l}=+\,\texttt{1}\,y\rangle\}\\ &\quad\{\!|\,\langle\texttt{23},\texttt{3}\rangle,\langle\texttt{42},\texttt{4}\rangle,\langle\texttt{23},\texttt{2}\rangle,\langle\texttt{23},\texttt{8}\rangle\,|\!\}\\ =\ &\{\!|\,\langle\texttt{23},\langle\texttt{s}=\texttt{13},\texttt{l}=\texttt{3}\rangle\rangle,\langle\texttt{42},\langle\texttt{s}=\texttt{4},\texttt{l}=\texttt{1}\rangle\rangle\,|\!\}\end{aligned}$$

Next to unary grouping, we include binary grouping with the relational groupjoin operator $⟗\{\}$ as defined by Moerkotte and Neumann [MN11]. We slightly deviate from their definition by applying the aggregate function to pairs of values from both operands. Groupjoin is not supported by SQL:2003 and most relational query engines. However, it can be simulated with a left outer join using equivalences by Moerkotte and Neumann.

As the only order-aware operator in $\mathcal{MA}$, `#{}` provides an equivalent to SQL's window function `row_number()`. In `#{`s_p`,`s_o`}`, scalar function s_p returns a partitioning key for each element of the input multiset. Elements of each partitions are sorted according to the *sorting key* provided by s_o and enumerated starting with `1`. Consider the following example that partitions

$\mathcal{MA}$-ty-table

$$\frac{\Sigma(t) = \langle \ell_1 : \pi_1, \ldots, \ell_n : \pi_n \rangle}{\text{DB}_t : \{\!\!\{ \langle \ell_1 : \pi_1, \ldots, \ell_n : \pi_n \rangle \}\!\!\}}$$

$\mathcal{MA}$-ty-lit

$$\frac{[\vdash v_i : \pi]_{i=1}^{n}}{\{\!\!\{ v_1, \ldots, v_n \}\!\!\} : \{\!\!\{ \pi \}\!\!\}}$$

$\mathcal{MA}$-ty-project

$$\frac{q : \{\!\!\{ \delta \}\!\!\} \qquad \vdash s : \delta \to \delta'}{\pi\{s\}\ q : \{\!\!\{ \delta' \}\!\!\}}$$

$\mathcal{MA}$-ty-select

$$\frac{q : \{\!\!\{ \delta \}\!\!\} \qquad \vdash s : \delta \to \texttt{Bool}}{\sigma\{s\}\ q : \{\!\!\{ \delta \}\!\!\}}$$

$\mathcal{MA}$-ty-union

$$\frac{q_1 : \{\!\!\{ \delta \}\!\!\} \qquad q_2 : \{\!\!\{ \delta \}\!\!\}}{q_1 \cup q_2 : \{\!\!\{ \delta \}\!\!\}}$$

$\mathcal{MA}$-ty-product

$$\frac{q_1 : \{\!\!\{ \delta_1 \}\!\!\} \qquad q_2 : \{\!\!\{ \delta_2 \}\!\!\}}{q_1 \times q_2 : \{\!\!\{ \langle \delta_1, \delta_2 \rangle \}\!\!\}}$$

$\mathcal{MA}$-ty-thetajoin

$$\frac{q_1 : \{\!\!\{ \delta_1 \}\!\!\} \qquad q_2 : \{\!\!\{ \delta_2 \}\!\!\} \qquad \vdash s : \delta_1 \to \delta_2 \to \texttt{Bool}}{q_1 \bowtie\{s\}\, q_2 : \{\!\!\{ \langle \delta_1, \delta_2 \rangle \}\!\!\}}$$

$\mathcal{MA}$-ty-semijoin

$$\frac{q_1 : \{\!\!\{ \delta_1 \}\!\!\} \qquad q_2 : \{\!\!\{ \delta_2 \}\!\!\} \qquad \vdash s : \delta_1 \to \delta_2 \to \texttt{Bool}}{q_1 \ltimes \{s\}\, q_2 : \{\!\!\{ \delta_1 \}\!\!\}}$$

$\mathcal{MA}$-ty-antijoin

$$\frac{q_1 : \{\!\!\{ \delta_1 \}\!\!\} \qquad q_2 : \{\!\!\{ \delta_2 \}\!\!\} \qquad \vdash s : \delta_1 \to \delta_2 \to \texttt{Bool}}{q_1 \rhd \{s\}\, q_2 : \{\!\!\{ \delta_1 \}\!\!\}}$$

$\mathcal{MA}$-ty-distinct

$$\frac{q : \{\!\!\{ \delta \}\!\!\}}{\delta\ q : \{\!\!\{ \delta \}\!\!\}}$$

$\mathcal{MA}$-ty-rownum

$$\frac{q : \{\!\!\{ \delta \}\!\!\} \qquad \vdash s_p : \delta \to \delta' \qquad \vdash s_o : \delta \to \delta''}{\#\{s_p, s_o\}\ q : \{\!\!\{ \langle \delta, \texttt{Int} \rangle \}\!\!\}}$$

$\mathcal{MA}$-ty-group

$$\frac{q : \{\!\!\{ \delta \}\!\!\} \qquad \vdash s_g : \delta \to \delta_g \qquad \vdash s_z : \delta_a \qquad \vdash s_f : \delta \to \delta_a \to \delta_a}{\Gamma\{s_g, s_z, s_f\}\ v : \{\!\!\{ \langle \delta_g, \delta_a \rangle \}\!\!\}}$$

rel-ty-outerjoin

$$\frac{\begin{array}{c} q_1 : \{\!\!\{ \delta_1 \}\!\!\} \\ q_2 : \{\!\!\{ \delta_2 \}\!\!\} \qquad \vdash s_p : \delta_1 \to \delta_2 \to \texttt{Bool} \qquad \vdash s_z : \delta_3 \qquad \vdash s_r : \delta_2 \to \delta_3 \end{array}}{q_1 ⟕ \{s_p, s_z, s_r\}\, q_2 : \{\!\!\{ \langle \delta_1, \delta_3 \rangle \}\!\!\}}$$

rel-ty-groupjoin

$$\frac{\begin{array}{c} q_1 : \{\!\!\{ \delta_1 \}\!\!\} \qquad q_2 : \{\!\!\{ \delta_2 \}\!\!\} \\ \vdash s_p : \delta_1 \to \delta_2 \to \texttt{Bool} \qquad \vdash s_z : \delta_a \qquad \vdash s_f : \langle \delta_1, \delta_2 \rangle \to \delta_a \to \delta_a \end{array}}{q_1 \,\Gamma\!\!\bowtie \{s_p, s_z, s_f\}\, q_2 : \{\!\!\{ \langle \delta_1, \delta_a \rangle \}\!\!\}}$$

Figure 53: Typing rules for $\mathcal{MA}$ multiset operators.

$$
\begin{aligned}
\mathcal{M}[\![\text{⛁}_t]\!] &= [\![t]\!] \\
\mathcal{M}[\![\{\!\{v_1, \ldots, v_n\}\!\}]\!] &= [[\![v_1]\!], \ldots, [\![v_n]\!]] \\
\mathcal{M}[\![\pi\{s\}\ q]\!] &= [\ [\![s]\!]\ x \mid x \leftarrow \mathcal{M}[\![q]\!]\ ] \\
\mathcal{M}[\![\sigma\{s\}\ q]\!] &= [\ x \mid x \leftarrow \mathcal{M}[\![q]\!], [\![s]\!]\ x\ ] \\
\mathcal{M}[\![q_1 \cup q_2]\!] &= \mathcal{M}[\![q_1]\!] \mathbin{+\!\!+} \mathcal{M}[\![q_2]\!] \\
\mathcal{M}[\![\delta\ q]\!] &= \mathsf{nubWith}\ (\lambda x.x)\ \mathcal{M}[\![q]\!] \\
\mathcal{M}[\![\#\{s_p, s_o\}\ q]\!] &= \mathsf{concat}\ [\ \mathsf{enum}\ (\mathsf{sortWith}\ [\![s_o]\!]\ g.2) \\
&\qquad\qquad \mid g \leftarrow \mathsf{groupWith}\ [\![s_p]\!]\ \mathcal{M}[\![q]\!]\ ] \\
\mathcal{M}[\![q_1 \times q_2]\!] &= [\ \langle x, y\rangle \mid x \leftarrow \mathcal{M}[\![q_1]\!], y \leftarrow \mathcal{M}[\![q_2]\!]\ ] \\
\mathcal{M}[\![q_1 \ltimes\!\rtimes \{s\}\ q_2]\!] &= [\ \langle x, y\rangle \mid x \leftarrow \mathcal{M}[\![q_1]\!], y \leftarrow \mathcal{M}[\![q_2]\!], [\![s]\!]\ x\ y\ ] \\
\mathcal{M}[\![q_1 \ltimes \{s\}\ q_2]\!] &= [\ x \mid x \leftarrow \mathcal{M}[\![q_1]\!], \mathsf{or}\ [\ [\![s]\!]\ x\ y \mid y \leftarrow \mathcal{M}[\![q_2]\!]\ ]\ ] \\
\mathcal{M}[\![q_1 \rhd \{s\}\ q_2]\!] &= [\ x \mid x \leftarrow \mathcal{M}[\![q_1]\!], \mathsf{and}\ [\ \neg([\![s]\!]\ x\ y) \mid y \leftarrow \mathcal{M}[\![q_2]\!]\ ]\ ] \\
\mathcal{M}[\![\Gamma\{s_g, s_z, s_f\}\ q]\!] &= [\ \langle k, \mathsf{foldl}\ [\![s_f]\!]\ [\![s_z]\!]\ xs\rangle \\
&\qquad \mid \langle k, xs\rangle \leftarrow \mathsf{groupWith}\ [\![s_g]\!]\ \mathcal{M}[\![q]\!]\ ] \\
\mathcal{M}[\![q_1 \bowtie\!\rtimes \{s_p, s_z, s_r\}\ q_2]\!] &= [\ \langle x.1, [\![s_r]\!]\ x.2\rangle \mid x \leftarrow \mathcal{M}[\![q_1 \ltimes\!\rtimes \{s_p\}\ q_2]\!]\ ] \\
&\qquad \mathbin{+\!\!+} \\
&\qquad [\ \langle x, [\![s_z]\!]\rangle \mid x \leftarrow \mathcal{M}[\![q_1 \rhd \{s_p\}\ q_2]\!]\ ] \\
\mathcal{M}[\![q_1 \ltimes\!\rtimes\!\!\!\!\!\!\bowtie \{s_p, s_z, s_f\}\ q_2]\!] &= [\ \langle x, \mathsf{foldl}\ [\![s_f]\!]\ [\![s_z]\!]\ [\ y \mid y \leftarrow \mathcal{M}[\![q_2]\!], [\![s_p]\!]\ x\ y\ ]\rangle \\
&\qquad \mid x \leftarrow \mathcal{M}[\![q_1]\!]\ ]
\end{aligned}
$$

Figure 54: List interpretation of $\mathcal{MA}$ operators.

a collection of pairs by their first component and sorts according to the second component:

$$
\begin{aligned}
&\#\{\lambda x.x.1, \lambda x.x.2\}\ \{\!\{\langle 23, 1\rangle, \langle 23, 3\rangle, \langle 42, 8\rangle, \langle 42, 3\rangle, \langle 23, 3\rangle\}\!\} \\
&= \{\!\{\langle\langle 23, 1\rangle, 1\rangle, \langle\langle 23, 3\rangle, 3\rangle, \langle\langle 23, 3\rangle, 2\rangle, \langle\langle 42, 8\rangle, 2\rangle, \langle\langle 42, 3\rangle, 1\rangle\}\!\}
\end{aligned}
$$

Note that in the partition 23 a tie exists between two elements with sorting key 3. Recall that sortWith provides stable sorting such that the original order of tied elements is preserved. An implementation in SQL:2003 with `row_number()` however, will choose a random, non-deterministic order. This gap is not relevant for us: the lowering of $\mathcal{SL}$ to $\mathcal{MA}$ ensures that sorting keys for #{} are always unique.

6.2 GENERATING MULTISET PLANS

Before delving into the details of the $\mathcal{MA}$ code generator, we discuss the unordered representation of segment vectors and sketch the implementation of vector operators.

6.2.1 *Multiset Representation of Segment Vectors*

In Section 4.3, we describe the segment vector model based on indexes. Indexes align vector elements from the same iteration and link pairs of corre-

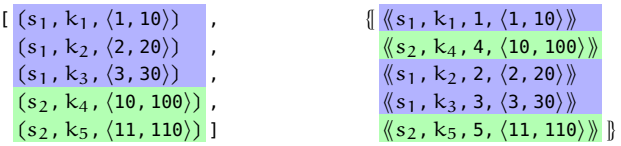


Figure 55: A vector with two segments and its unordered representation: a multiset of (nested) records.

sponding outer and inner vectors to represent nested lists. Segment vectors $\mathrm{D}\ (\delta_s, \delta_k, \delta_p)$ are interpreted as lists of type

$$[\langle \mathtt{s}:\delta_s, \mathtt{k}:\delta_k, \mathtt{p}:\delta_p \rangle]$$

with δ_s and δ_k denoting the types of outer and inner indexes, respectively, and δ_p denoting the payload type.

We lower list-based vectors directly to unordered multisets. To encode the order of list elements, each element is extended with a *order label*. Order labels encode the order of vector elements explicitly. Hence, a segment vector is lowered to the following multiset type:

$$\{\!\{ \langle \mathtt{s}:\delta_s, \mathtt{k}:\delta_k, \mathtt{o}:\delta_o, \mathtt{p}:\delta_p \rangle \}\!\}$$

It will be convenient to abbreviate the corresponding record constructor as follows:

$$\langle \mathtt{s}=s, \mathtt{k}=k, \mathtt{o}=o, \mathtt{p}=p \rangle \equiv \langle\!\langle s, k, o, p \rangle\!\rangle$$

An example for the multiset representation of a segment vector is shown in Figure 55.

In Section 4.3.2, we distinguish four forms of index transformations that propagate index changes from outer and inner vectors. All four transformations map to multisets in an obvious way:

- Replication and rekeying maps of type $\mathrm{R}\,r_1\,r_2$ and $\mathrm{K}\,r_1\,r_2$ are implemented as multisets of type $\{\!\{\langle \mathtt{f}:r_1, \mathtt{t}:r_2 \rangle\}\!\}$.
- Sorting and filtering transformations of type $\mathrm{S}\,r$ and $\mathrm{F}\,r$ are multisets of type $\{\!\{r\}\!\}$.

In Section 4.3, we have used abstract indexes. The concrete encoding of indexes is not relevant for the flattening translation and there are multiple indexing schemes to consider. For now, we will keep the representation of indexes as well as of order abstract and sketch the implementation of vector operators first. Section 6.2.2 demonstrates that the per-segment nature of vector operators maps naturally to $\mathcal{MA}$ operators. In Section 6.2.3, we make the vector encoding concrete and consider *natural* and *synthetic* indexing schemes as well as *eager* and *lazy* encodings of order.

6.2.2 *Implementing Segment Operators*

$\mathcal{MA}$ operators process their complete input collection uniformly. In contrast, $\mathcal{SL}$ segment operators have to observe the segment structure of their inputs. Our discussion of $\mathcal{SL}$ operators semantics hints towards the relational implementation of these operators, in particular segment joins (Section 5.4.4).

Here, we demonstrate that $\mathcal{SL}$ operators indeed map to idiomatic and efficient relational plans.

Each vector operator is expanded into a small number of $\mathcal{MA}$ operators. For some vector operators there is a trivial one-to-one mapping to a single $\mathcal{MA}$ operator (*i.e.* π). Others are centered around one particular operator (typically join operators) and use additional projections to maintain administrative information (indexes and order). In the following, we sketch the $\mathcal{MA}$ implementation of the five vector operators `projectV{}`, `reduceV{}`, `semijoinV{}`, `nestjoinV{}` and `alignV`. We postpone the discussion of order representation and use the placeholder $\square$ for order values in records.

- The vector operator `projectV{}` expresses a data-parallel scalar computation. As it deals with individual vector elements only, `projectV{}` is oblivious to the input vector's segment structure. Indexes and the order of elements do not change and need to be preserved from the operator's input. A vector projection `projectV{λx.+x.1x.2}` maps to a $\mathcal{MA}$ projection π`{λx.⟪x.s, x.k, x.o, +x.p.1x.p.2⟫}` which applies the scalar expression to the payload `x.p` and copies indexes and order without modification.

- Operator `reduceV{}` folds each segment individually to implement data-parallel aggregates. As each element in the multiset encoding is annotated with the outer index, `reduceV{}` maps directly to grouped aggregation. Consider the example in Figure 56 where `reduceV{}` computes the sum of each segment. By including segment identifiers as the grouping key (marked red), it can be implemented by Γ{}. A subsequent projection establishes the complete multiset vector encoding. The inner index can be derived from the outer index because we statically know that each folded segment will have exactly one element. Note that this choice of inner index is in line with the typing rule for the `reduceV{}` operator in Figure 38 which requires the result vector to use the same type of outer and inner indexes.

- Vector join operators like `semijoinV{}` observe the segment structure of their inputs and join corresponding segments individually. As demonstrated in Figure 57, adding a comparison of segment identifiers to the join predicate is sufficient to implement this behaviour. The comparison ensures that only elements from corresponding segments are considered for the actual join predicate. As `semijoinV{}` only removes elements from its left operand, the inner index of the left operand stays intact.

- Vector operator `nestjoinV{}` creates a segment for each element of the left operand. Each segment in the result vector contains all matching elements from the corresponding segment of the right operand. In Figure 58 we sketch the $\mathcal{MA}$ implementation of `nestjoinV{}`. Operator $\bowtie${} evaluates the join predicate and combines matching tuples. As for `semijoinV{}`, we add a comparison of segment identifiers to the join predicate to align elements from corresponding segments only. After the join, a projection utilizes the left operands' inner index as outer index in the result. Unique inner indexes for the result are obtained by pairing inner indexes of the operands.

- Operator `alignV` pairs corresponding elements from two vectors. It maps to a join

  $\bowtie${λx y.=x.k y.k}

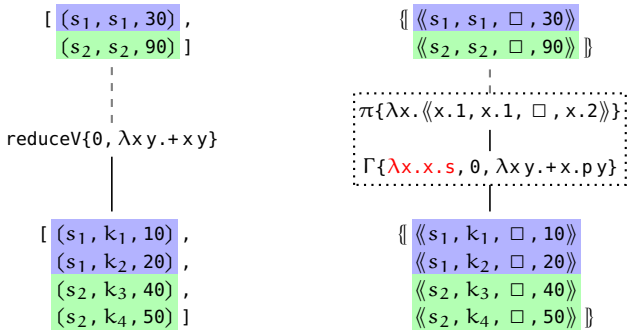


Figure 56: $\mathcal{MA}$ implementation of `reduceV{}` (sketch).

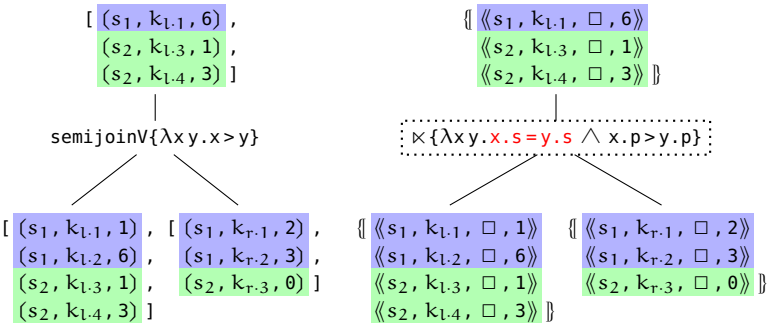


Figure 57: $\mathcal{MA}$ implementation of `semijoinV{}` (sketch).

on inner indexes. Again, the typing rule for `alignV` in Figure 38 guarantees that both operands have inner indexes of the same type. As we know that the vector shape of both operands is identical, a projection chooses the outer index of the left operand to establish the proper vector encoding.

All operators discussed exhibit a rather direct and idiomatic $\mathcal{MA}$ implementation. Although we have deliberately chosen simple operators here, the detailed discussion in Section 6.3 will show that the other operators follow the same pattern: Per-element and administrative operators map to projec-

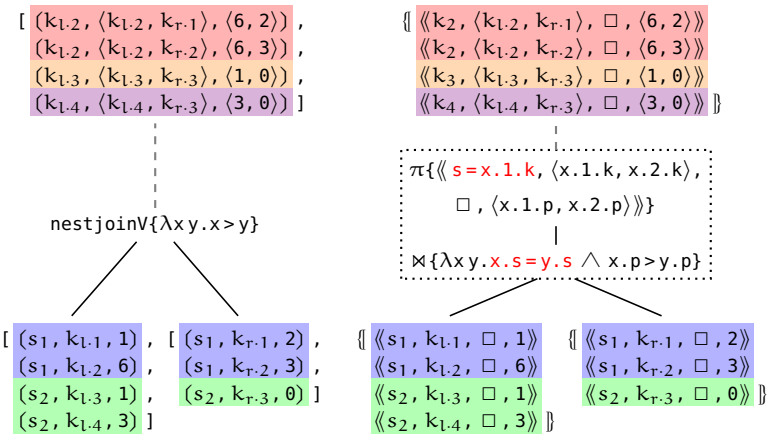


Figure 58: $\mathcal{MA}$ implementation of `nestjoinV{}` (sketch).

tions and simple joins, whereas data-parallel per-segment operators extend join predicates and the like with index comparisons.

6.2.3 *Vector Encoding: Indexes and Order*

To define a concrete multiset encoding of segment vectors, we discuss indexing schemes (Section 6.2.3.1) and order representation (Section 6.2.3.2).

In *Loop-Lifting* (Section 2.2.2), indexes and order are entwined: in *Loop-Lifting*'s relational encoding, each tuple is annotated with its relative list position. Iteration identifiers, in turn, are derived from list positions and serve as the basis for indexes that link outer and inner relations. In our approach, index and order representation are related as well: both indexes and the order of vector elements are initially derived from primary keys of base tables. The segment vector model, however, enables us to consider those aspects separately: the index maintenance of vector operators is defined without prescribing a physical or logical representation of the order of vector elements.

Separating the two topics is beneficial for two reasons. First, the behaviour of vector operators is not symmetrical with regard to index and order maintenance. Certain operators (*e.g.* `repV`, `nestjoinV{}`) require to recompute indexes but allow to preserve the order representation. Others (most prominently `sortV{}`) change the order of elements but leave indexes intact. We aim to exploit these asymmetries in the translation to relational algebra. In particular, we describe an implementation of the `sortV{}` operator that, effectively, has no runtime cost. Second, the separation enables an adaption of *Query Flattening* to unordered collections that require indexes but no order labels (see Section 9.2).

6.2.3.1 *Synthetic and Natural Indexes*

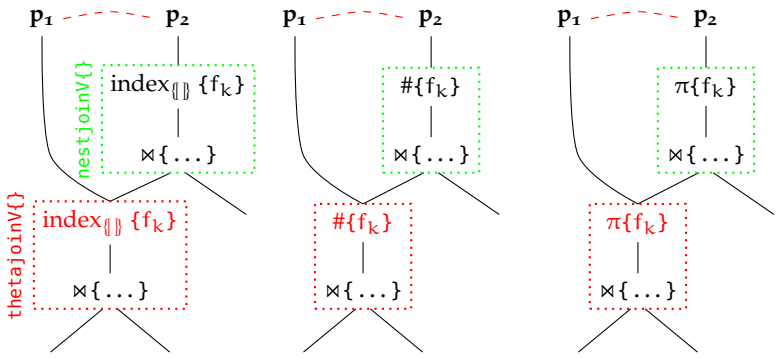
Indexes are derived from the primary keys of base tables. The implementation of a $\mathcal{SL}$ operator has to recompute indexes whenever the inner indexes of the input vectors do not uniquely identify elements of the result. Concerned are nesting operators `groupV{}` and `repV` as well as the join operators `thetajoinV{}` and `nestjoinV{}`. These operators take a prominent place in most queries. An efficient implementation of index computation is paramount.

The list interpretation of $\mathcal{SL}$ operators derives new indexes by composing indexes of operands. For `thetajoinV{}`, for example, Equation ($\mathcal{SL}$-THETAJOIN), provides unique indexes for the result vector as pairs of old indexes of both operands. In this section, we discuss the viability of this indexing scheme for the $\mathcal{MA}$ interpretation of $\mathcal{SL}$ and compare it with an alternative indexing scheme that has been used in related work. We discuss two kinds of indexes:

1. *Synthetic indexes* are uniform integer indexes derived by enumerating vector elements.

2. *Natural indexes* are non-uniform tuples of base table keys.

For the purpose of the discussion, we express abstract index computation with the $\mathcal{MA}$ operator $\text{index}_{\{\!|\,|\!\}}\,\{s\}$. Applied to a multiset of type $\{\!|\delta|\!\}$, the operator extends each element with an index value of some type δ_k and returns a multiset of type $\{\!|\langle \delta, \delta_k \rangle|\!\}$. Index values are derived from the result of expression s. For example, $\text{index}_{\{\!|\,|\!\}}\,\{\lambda x.x.k\}$ computes new indexes



(a) Abstract Index (b) Synthetic Index (c) Natural Index

Figure 59: Join plan sketches with index computation instantiated with synthetic and natural indexes. $f_k \equiv \lambda x.\langle x.1.k, x.2.k\rangle$

from the inner index of the input. Consider the plan fragment in Figure 59a. It sketches a relational plan implementing `thetajoinV{}` and `nestjoinV{}`. For both operators, indexes for the join result are derived from the combination of indexes of both operands.

Index computation has to be *deterministic*. This property is crucial to construct nested results from vectors that are linked by indexes. In Figure 59a, the index computed as part of the `thetajoinV{}` implementation is shared in the relational plans p_1 and p_2 and links the corresponding outer and inner vectors. We do not prescribe a particular evaluation strategy for relational plans with sharing (we discuss evaluation strategies for shared subplans in Section 6.5). While the shared sub-plan's result might be physically materialized and reused, the plan might also be evaluated by unfolding it into two separate backend queries which are evaluated independently. Without deterministic index computation, the relationship between outer and inner vector could not be established.

SYNTHETIC INDEXES Both *Loop-Lifting Query Shredding* (Section 2.2) are based on *synthetic indexes* (coined *flat indexes* by Cheney *et al.* [CLW14a]). The idea is simple: Sort the input bag in some order, enumerate its elements and use the enumeration as the index values. To ensure a deterministic index, the input bag is sorted by a key. In $\mathcal{MA}$, synthetic indexes can be computed with `#{}`:

$$\text{index}_{⟦⟧}\ \{s\}\ q = \#\{\lambda x.\langle\rangle, s\}\ q$$

In Figure 59b, synthetic indexes for join results are computed based on the unique combination of the left and right inputs' indexes. The synthetic index scheme combines keys by sorting and enumerating according to both indexes:

$$\#\{\lambda x.\langle\rangle, \lambda x.\langle x.1.k, x.2.k\rangle\}$$

A synthetic index value computed in this way is a single integer value. Hence, synthetic index values are compact and can be efficiently compared. Synthetic indexes provide for cheap sorting and hashing in physical operators. *Creating* a synthetic index based on row numbering, however, is not cheap. In relational query engines based on unordered collections, numbering operators are evaluated by sorting the input. Sorting is an expensive operation. Furthermore, sorting is a *blocking* operation that prevents pipelining

and materializes intermediate results. $\mathcal{MA}$ plans derived from $\mathcal{SL}$ queries with synthetic indexes are littered with expensive sorting operations.

In some cases, numbering operators can be simplified or eliminated entirely. Rittinger [Rit11] describes a set of algebraic rewrites that aim to minimize numbering operators (Section 2.2.2). Such rewrites, however, do not provide a guarantee to eliminate all numbering operators. Index-generating numbering operators are hard to remove in the presence of nested data, *i.e.* outer and inner vectors linked by indexes as in Figure 59. In Figure 59b, both plans p_1 and p_2 depend on the index values generated by the lower `#{}` operator. This operator can not be replaced with a local rewrite — index values need to be consistent across both plans.

Next, we consider an indexing scheme that does not introduce expensive operations that have to be removed afterwards.

NATURAL INDEXES Rather than computing integer surrogates from table keys and use those surrogates as indexes, we can use base table keys themselves as indexes. We have already employed this scheme in the indexed semantics of $\mathcal{FL}$ (Section 4.2.4) and the semantics of $\mathcal{SL}$ (Section 4.3.2.2 and Section 5.4.4). We follow Cheney *et al.* [CLW14a] and call such indexes *natural indexes*. In $\mathcal{MA}$, natural indexes are obtained as follows:

$$\text{index}_{\{\!|\,|\!\}}\ \texttt{\{s\}}\ \texttt{q} = \pi\texttt{\{}\lambda\texttt{x.}\langle\texttt{x, s x}\rangle\texttt{\}}\ \texttt{q}$$

Under this indexing scheme, indexes are non-uniform: In general, two vectors will have different types for their outer and inner indexes, respectively. However, in a type-correct $\mathcal{SL}$ program obtained by *Query Flattening*, vectors have the same type of indexes in all situations in which indexes are actually compared (Figure 38):

1. For a pair of an outer vector V_o and an inner vector V_i, the type of the outer index of V_i matches the type of the inner index of V_o.
2. Vectors that are combined with `alignV` have the same vector shape and therefore the same type of outer and inner index.

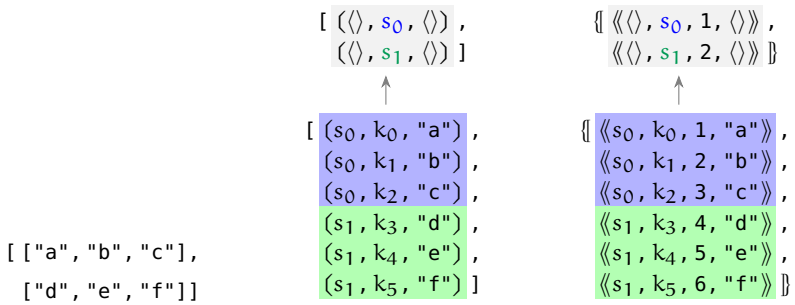
In Figure 59c, natural indexes are used. Indexes are combined simply by forming pairs of input indexes:

$$\pi\texttt{\{}\lambda\texttt{x.}\langle\texttt{x, }\langle\texttt{x.1.k, x.2.k}\rangle\rangle\texttt{\}}$$

Let δ_{k_1}, δ_{k_2} and δ_{k_3} denote the inner index types of the three input relations. Then the indexes generated for the lower (`thetajoinV{}`) and upper (`nestjoinV{}`) vector operators are of type $\langle\delta_{k_1}, \delta_{k_2}\rangle$ and $\langle\langle\delta_{k_1}, \delta_{k_2}\rangle, \delta_{k_3}\rangle$, respectively.

In general, natural indexes are composed from nested records of scalar data, most often consisting of primary key attributes of base tables. Comparing them is more expensive than comparing single integer values. Therefore, we expect that comparisons on natural indexes have higher cost than on synthetic indexes. However, we assume that the cost of sorting induced by synthetic indexes is considerably higher.

Under the natural indexing scheme, a large class of queries can be implemented without any numbering operators for index maintenance. Indeed, Section 6.3 shows that `appendV` is the only operator that requires a numbering operator for index maintenance.



(a) Nested list. (b) Segment vectors. (c) Multiset.

Figure 60: A nested list represented as a pair of segment vectors and a pair of multisets. Order labels on multiset elements (here of type `Int`) describe the order of segment elements.

6.2.3.2 *Eager and Lazy Order*

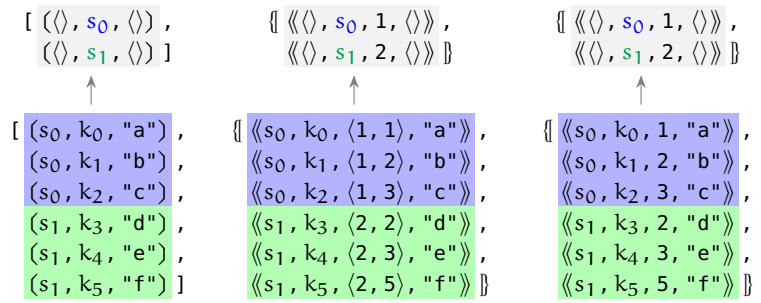
All languages defined in query flattening are based on lists, *i.e.* ordered collections. We have defined the semantics of $\mathcal{SL}$ operators based on flat lists.

Query Flattening uses intermediate languages ($\mathcal{CL}_d$, $\mathcal{FL}$, $\mathcal{SL}$) centered around ordered lists. Here, however, we lower $\mathcal{SL}$ operators to $\mathcal{MA}$ operators on unordered multisets. Hence, we have to define an explicit encoding of ordered vectors in terms of multisets. Although some work can not be avoided to maintain order information, the cost of order maintenance should be minimized. In this section, we make observations that lead to an efficient unordered backend. Same as for the index scheme, our aim is to minimize the effort for order maintenance upfront *by construction* instead of relying on subsequent optimizations.

ORDER LABELS An ordered list $[x_1, \ldots, x_n]$ can be represented in terms of an unordered multiset by encoding order as data: Every list element x is equipped with an *order label* o such that for two list elements x_i, x_j, we have $o_i < o_j$ if and only if $i < j$. Then, instead of the original list, the multiset $\{\!\{\langle x_1, o_1\rangle, \ldots, \langle x_n, o_n\rangle\}\!\}$ can be stored. The original list can be obtained by sorting the multiset on the order labels. List order then is defined through the order relation on order labels.

A list in a $\mathcal{CL}$ query maps to a segment in a $\mathcal{SL}$ vector. To maintain the order of $\mathcal{CL}$ list elements, we have to maintain the order of elements in the corresponding segment. Hence, when lowering a segment vector to a multiset, we add order labels such that for each segment, labels impose a total order on the elements of that segment. In Figure 60, the order of the two elements of the outer list is described by order labels on the multiset that encodes the outer vector. The order of elements of the two inner lists is described by order labels on the segments s_0 and s_1 on the multiset that encodes the inner vector.

$\mathcal{CL}$ lists are originally created from unordered base tables. As defined in Section 1.4, base tables are interpreted as lists in the order of their primary key. Primary keys of base tables serve as the basis for order labels. An $\mathcal{MA}$ implementation of an $\mathcal{SL}$ operator has to maintain order labels such that for each segment of the result, order labels describe the total order of segment elements.



(a) Segment vectors. (b) Multiset. (c) Multiset.

Figure 61: Order encoding with and without ordering of segments.

ORDER OF SEGMENTS The list semantics of $\mathcal{SL}$ operators (Section 4.3.2.2) not only maintains the order of segment elements but also the order of segments. The order of segments in an inner vector is the same as the order of the corresponding elements in the outer vector.

This consistent order of segments can be exploited by a backend with ordered collections (Section 4.3.2.3). For the $\mathcal{MA}$ backend, though, maintaining this property incurs unnecessary overhead. The relationship between elements of an outer vector V_o and segments in an inner vector V_i is described by indexes. To relate each element of V_o with the corresponding segment in V_i, the order of segments in V_i is not relevant. In Figure 61b, order labels are pairs that not only describe the order of segment elements but also the order of segments that corresponds to the order of elements in the outer vector. In Figure 61c, on the other hand, order labels only describe the order of elements relative to a segment. Although the latter representation is more liberal, it still faithfully represents the original nested list.

PROPAGATING ORDER CHANGES Since we don't insist on ordering segments, order labels become a local property of one particular multiset. If the order of elements in an outer vector changes (*e.g.* because of `sortV{}`), the change in order labels does not have to be propagated to multisets that encode any inner vectors.

We illustrate this with an example. Consider the expression $\texttt{sort}^{\uparrow}\ e$, where e is an expression of type $\texttt{[[}\langle\texttt{[Int],Text}\rangle\texttt{]]}$. By shredding we obtain for e the package $\texttt{[[}\langle\texttt{[Int]}^{V_n}\texttt{,Text}\rangle\texttt{]}^{V_i}\texttt{]}^{V_o}$ where the segments of vector V_i encode the lists that are to be sorted. Sorting is implemented with the corresponding vector operator applied to V_i. By SHRED-SORT-LIFT we have

$$\texttt{sort}^{\uparrow}\ e \triangleright \texttt{[[}\langle\texttt{[Int]}^{V'_n}\texttt{,Text}\rangle\texttt{]}^{V'_i}\texttt{]}^{V_o}$$

and emit the vector program

$$\begin{aligned}(V_s\texttt{,I}) &\leftarrow \texttt{sortV\{}\lambda\texttt{x.x.2\}}\ V_i\\ V'_i &\leftarrow \texttt{projectV\{}\lambda\texttt{x.x.1\}}\ V_s\\ V'_n &\leftarrow \texttt{appsortV I}\ V_n\end{aligned}$$

By sorting V_i, elements of V_i that encode pairs of type $\langle\texttt{[Int],Text}\rangle$ are rearranged. We depict the situation with concrete arguments and integer order labels in Figure 62. In the multiset corresponding to vector V_i, this rearrangement is reflected in a change to the order labels. The order of elements of the innermost lists of type `[Int]`, however, is not affected and their order labels stay the same. Because the inner index of V_i does not

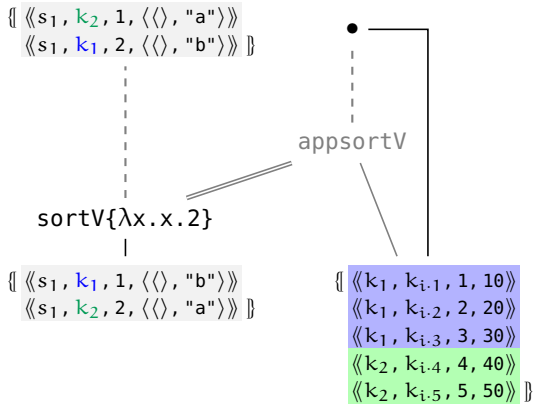


Figure 62: Maintaining the order of segments in inner vectors with segment-relative order labels involves no work. `sort`$^{\uparrow}$ `[[⟨[10,20,30],"b"⟩,⟨[40,50],"a"⟩]]`

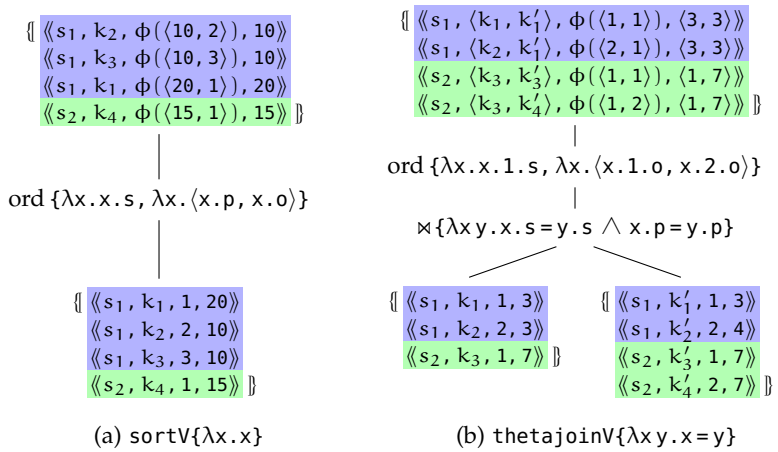
change through sorting, the inner vector V_n does not need to be changed at all. The multiset implementation of `appsortV` then returns the inner vector unmodified.

ABSTRACT ORDER LABELS Before we discuss concrete encodings of order labels, we discuss maintenance of order labels. For the purpose of the discussion, we use the abstract labeling function ord $\{s_s, s_o\}$ to extend a multiset with order labels. For each element of the input, s_o returns a value of some type δ_o that represents the elements' order. s_s returns the outer index of an element to enable ord `{}` to create segment-relative order labels. Given a multiset q of type $\{\!|\delta|\!\}$, ord $\{s_s, s_o\}$ q evaluates to a multiset of type $\{\!|\langle\delta, \phi(\delta_o)\rangle|\!\}$ where $\phi(\delta)$ denotes the type of order labels. We also use $\phi(x)$ to denote an abstract order label derived from value x.

Values of any scalar type (*i.e.* atomic values and records) can serve as the basis for order labels. The order relation defined on the scalar type determines the order condensed into labels. As defined in Section 1.4, the order relation on records is defined as the lexicographic order on record fields according to the order of record labels. Creating order labels from records — in particular pairs — *composes* orders: Order labels derived with ord $\{f_g, \lambda x.\langle s_1, s_2\rangle\}$ order elements primarily by expression s_1 and secondarily (for ties in s_1) by expression s_2.

MAINTAINING ORDER LABELS Most vector operators merely preserve the order of elements in their input (*e.g.* `projectV{}`). Some operators, however, have to explicitly *maintain* the order of elements: Either because the order actually changes (*e.g.* `sortV{}`) or because the order labels of one of the operators' inputs are not sufficient to uniquely describe the order of elements in the operators' result (*e.g.* `thetajoinV{}`). We discuss order label maintenance with three examples:

1. Vector operator `sortV{s}` has to recompute order labels as it changes the order of segment elements. Segment elements are rearranged based on the result of function s applied to the vector payload. In Figure 63a, `sortV{λx.x.p}` reorders elements according to the integer payload.



(a) `sortV{λx.x}` (b) `thetajoinV{λxy.x=y}`

Figure 63: Maintaining order labels for vector operators `sortV{}` (Figure 63a) and `thetajoinV{}` (Figure 63b)

New order labels are derived from the result of function s. However, as demonstrated in the example, ties can occur between elements of a segment. Ties are resolved by including the original order label as a secondary order criterion in the call to ord. This implements the stable sorting semantics defined for the $\mathcal{CL}$ `sort` combinator and its lifted variant `sort`$^{\uparrow}$.

2. List semantics of join combinators imply that all matches for a particular element of the left operand come before the matches for the elements' successors. The order of matches for one particular left element reflects the original order of elements in the right operand.

   As depicted in Figure 63b, this behaviour is easily implemented by composing the order labels of the left and right operand. Elements in the join result are primarily ordered according to the order labels of the left join operand and only secondarily according to the order labels of the right operand.

3. Operator `appendV` appends individual pairs of corresponding segments. It implements the list-based semantics of the corresponding combinator `append`$^{\uparrow}$. For each segment identifier s, elements from the left operands' segment s appear before elements from the right operands' segment s. At the same time, the original order of elements in segment s of the left and right operands, respectively, has to be preserved.

   In Figure 64 we sketch the multiset implementation of `appendV`. Order labels are recomputed for the individual operands based on their original order labels. Including integer tags `1` and `2`, respectively, ensures that elements from the left operand always have a smaller order label than elements from the right operand. A regular multiset union, then, is sufficient to implement `appendV`.

Note that in all three examples, we ensure that generated order labels are unique in a segment. New order labels are always computed based on a composition with the original labels: by pairing order labels that are unique in the respective input (`thetajoinV{}`), by using unique labels to resolve ties in the sorting key (`sortV{}`) and by prefixing labels with a unique tag before merging labels into one multiset (`appendV`).

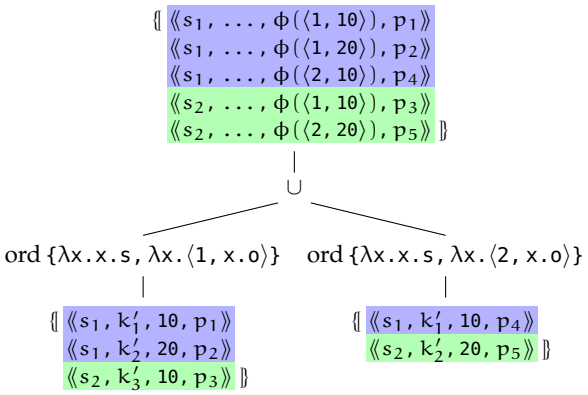
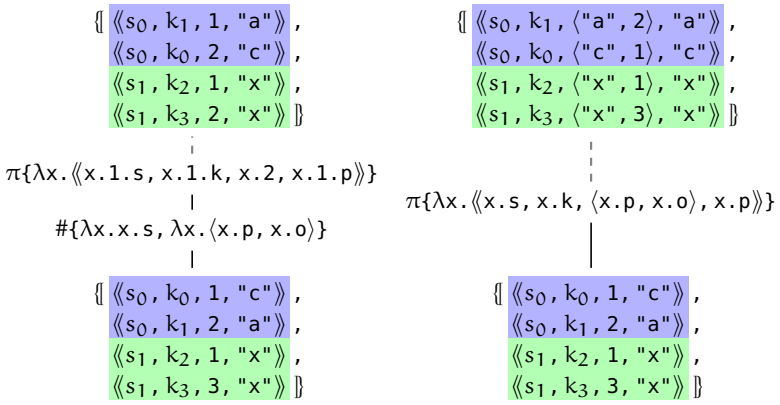


Figure 64: Maintaining order labels for vector operator `appendV`



(a) Eager. (b) Delayed.

Figure 65: Eager and delayed computation of order labels.

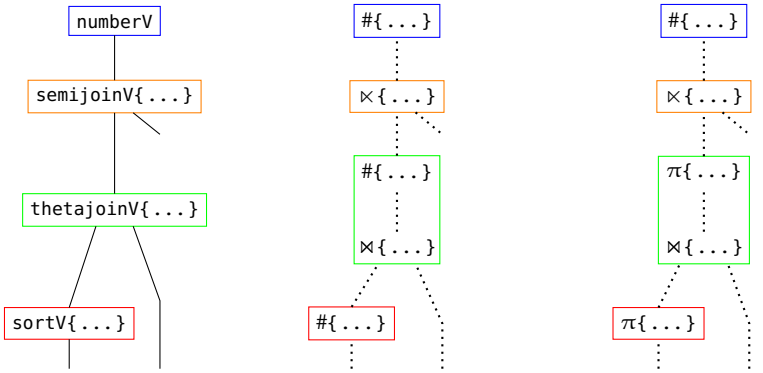
ENCODING ORDER LABELS The five operators discussed in the previous paragraph are the only ones that have to recompute order labels. Additionally, `tableV{}` has to compute initial order labels from base tables. All other operators can copy order labels from their inputs. Thus, the effort for implementing order-preserving $\mathcal{SL}$ operators on unordered multisets depends solely on the implementation of ord.

The alternatives to implement order label computation are the same as for index computation: either derive integer surrogates as order labels or use the underlying values that describe element order. The obvious choice is to use

$$\text{ord}\,\{s_s, s_o\}\ q = \#\{s_s, s_o\}\ q$$

to derive order labels of type `Int`. We obtain dense integer labels that enumerate the elements of each segment in the correct order. The resulting multiset algebra plans feature one numbering operator `#{}` for each $\mathcal{SL}$ operator that requires updating order labels. As an example, consider $\mathcal{SL}$ operator `sortV{λx.x}` that sorts its input vector by the payload field. In Figure 65a, this operator is implemented with dense integer order labels: numbering operator `#{}` derives new order labels and a subsequent projection rearranges record fields into the format expected for vector encodings.

The performance implications of plans littered with numbering operators have been discussed in Section 6.2.3.1. We can expect that sorting dominates



(a) Vector plan. (b) Eager labels. (c) Delayed labels.

Figure 66: Order encoding schemes.

the execution time and prohibits pipelining. Similar to the natural index scheme, however, we can avoid introducing numbering operators in the first place.

Order label computation is based on data that determines the order of segment elements. Recall that the original order of $\mathcal{CL}$ lists is induced by primary keys of base tables. Instead of condensing those keys into uniform integer labels, we preserve them. We use

$$\text{ord}\ \{f_s, f_o\}\ q = \pi\{\lambda x.\langle x, f_o\ x\rangle\}\ q$$

to derive order labels. Here, we essentially track the record fields that describe order. An example is given in Figure 65b. The order labels computed here merely combine the input payload of type `Text` (the primary sorting criterion) with the original order labels. The resulting pairs precisely describe the order of elements computed by `sortV{λx.x}`. The effort to evaluate `sortV{}` on multisets is minimal. We refer to this scheme for order labels as *lazy order* because any actual effort for list positions is delayed as long as possible. We only *materialize* list positions when absolutely necessary — for example when encountering a `numberV` operator.

Note that the resulting order labels are non-uniform: their type depends on the order-maintaining operators occurring in the plan. In general, two multisets that encode different segment vectors have different types of order labels. This does not pose a problem, however. Earlier in this section, we have established that order labels are local to a given vector. In $\mathcal{MA}$ plans derived from $\mathcal{SL}$ plans, we never compare order labels across vectors[1].

In Figure 66 we sketch order maintenance for an $\mathcal{SL}$ plan with both alternatives for order labels discussed here. In the $\mathcal{SL}$ plan in Figure 66a, operators `thetajoinV{}` and `sortV{}` require maintenance of order labels. Generating integer order labels with `#{}` (Figure 66b) requires three `#{}` operators. Next to one `#{}` generating list positions for `numberV`, two more are required to update order labels. The resulting plan encodes substantial sorting effort. In contrast, the plan in Figure 66c employs *lazy order* and features only one occurrence of `#{}`. Projections track the data that induces segment order up until the point where positions actually have to be computed by `numberV{}`. The plan encodes only minimal runtime effort for order maintenance in the form of projections.

1 With the exception of `appendV`. We elaborate on this problem in Section 6.3

6.3 GENERATING MULTISET PLANS

We now put the concepts discussed in Section 6.2 to work and describe the $\mathcal{MA}$ code generator based on natural indexes and lazy order. All essential ingredients are in place: the implementation of segment semantics as well as the computation of indexes and order labels. We specify translation rules for all vector operators based on these concepts.

We define an interpretation $\mathcal{R}[\![-]\!]_\rho$ of vector operators in terms of $\mathcal{MA}$ operators. It plugs directly into the semantics of vector programs defined in Section 4.3.2.2 and can be considered an alternative to the list interpretation of vector operators. Evaluating $\mathcal{R}[\![a]\!]_\rho$ maps the vector operator application a to $\mathcal{MA}$ expressions that evaluate to the multiset encoding of either a segment vector or an index transformation. The result of preceding vector applications is provided by environment ρ. As a simple example, operator `projectV{}` is lowered by Rule ($\mathcal{MA}$-PROJECT):

$$\mathcal{R}[\![\texttt{projectV\{s\}}\ V]\!]_\rho = \pi\{\lambda x.\langle\!\langle \texttt{x.s}, \texttt{x.k}, \texttt{x.o}, \texttt{s x}\rangle\!\rangle\}\ \rho(V) \quad (\mathcal{MA}\text{-PROJECT})$$

`projectV{f}` is translated given the multiset representation of its input vector obtained with $\rho(V)$. It maps to a single $\pi\{\}$ operator that preserves indexes and order labels while applying the scalar function f to the payload of the multiset vector encoding.

In the following rules, we often construct indexes, order labels etc. from pairs generated by a join operator. To abbreviate this pattern, we write

$$\text{pair}_\ell(e) = \langle e.1.\ell, e.2.\ell\rangle$$

In the construction of $\mathcal{MA}$ expressions, some translation rules re-use $\mathcal{MA}$ expressions (*e.g.* ($\mathcal{MA}$-GROUP)). This reflects the sharing in $\mathcal{SL}$ programs. If sharing of operator results is made explicit, we obtain DAG-shaped $\mathcal{MA}$ plans. We do not consider sharing in the translation rules. Standard implementation techniques (*e.g.* hash consing) can recover sharing and make it explicit in plans.

TABLE REFERENCES, LITERAL VECTORS Base tables and literal lists are handled by rules ($\mathcal{MA}$-LIT) and ($\mathcal{MA}$-TABLE).

$$\mathcal{R}[\![\texttt{litV\{[}v_1,\ldots,v_n\texttt{]\}}]\!]_\rho = \{\!|\langle\!\langle\langle\rangle, \texttt{i}, \texttt{i}, v_i\rangle\!\rangle_{i=1}^n|\!\} \quad (\mathcal{MA}\text{-LIT})$$

$$\mathcal{R}[\![\texttt{tableV\{t\}}]\!]_\rho = \pi\{\lambda x.\langle\!\langle\langle\rangle, \text{pk}_t(\texttt{x}), \text{pk}_t(\texttt{x}), \texttt{x}\rangle\!\rangle\}\ (\text{⛁}\, t) \quad (\mathcal{MA}\text{-TABLE})$$

Both have to provide inner indexes as well as order labels. As discussed previously, ($\mathcal{MA}$-TABLE) derives those from the primary key value $\text{pk}_t(r)$ of a base table row r. For literal tables, we use list positions to index elements. Note that both rules put all elements into the top-level unit segment by choosing the constant $\langle\rangle$ as outer index.

SEGMENT OPERATORS $\mathcal{MA}$ implementations of operators `selectV{}` and `numberV` are independent of the index and order label scheme. Both map directly to their $\mathcal{MA}$ counterparts.

$$\mathcal{R}[\![\texttt{selectV\{s\}}\ V]\!]_\rho = (q_s, q_m) \quad (\mathcal{MA}\text{-SELECT})$$

$$\text{where} \quad q_s \equiv \sigma\{\lambda x.\texttt{s x.p}\}\ \rho(V)$$

$$q_m \equiv \pi\{\lambda x.\texttt{x.k}\}\ q_s$$

Vector selection is implemented by σ{} with indexes, order labels and payload being preserved from the input. A segment map is created from the result of q_s that contains only those indexes surviving the selection and eliminates stale segments in any inner vectors.

$$\mathcal{R}[\![\texttt{numberV}\ V]\!]\rho = q_d \qquad (\mathcal{MA}\text{-NUMBER})$$
$$\text{where}\quad q_n \equiv \#\{\lambda x.\texttt{x.s}, \lambda x.\texttt{x.o}\}\ \rho(V)$$
$$q_d \equiv \pi\{\lambda x.\langle\!\langle \texttt{x.1.s}, \texttt{x.1.k}, \texttt{x.1.o}, \langle \texttt{x.1.p}, \texttt{x.2}\rangle\rangle\!\rangle\}\ q_s$$

In rule ($\mathcal{MA}$-NUMBER), segment positions are created with #{} by partitioning on outer indexes and enumerating each partition in the order dictated by the order labels.

$$\mathcal{R}[\![\texttt{sortV\{s\}}\ V]\!]\rho = (q_s, q_m) \qquad (\mathcal{MA}\text{-SORT})$$
$$\text{where}\quad q_s \equiv \pi\{\lambda x.\langle\!\langle \texttt{x.s}, \texttt{x.k}, \langle \texttt{s x.p}, \texttt{x.o}\rangle, \texttt{x.p}\rangle\!\rangle\}\ \rho(V)$$
$$q_m \equiv \pi\{\lambda x.\texttt{x.k}\}\ q_s$$

Rule ($\mathcal{MA}$-SORT) delays any actual sorting and simply records the sorting key as a prefix to the original order labels. As discussed in Section 6.2.3.2, by preserving the original order labels as a secondary ordering criterion, we implement stable sorting. A sorting transformation q_m is generated by ($\mathcal{MA}$-SORT) to comply with the type of the `sortV{}` vector operator. As the multiset representation of vectors does not depend on the order of segments, it will not actually be applied to inner vectors (see rule ($\mathcal{MA}$-APPSORT)).

To eliminate duplicates, $\mathcal{MA}$ provides the δ operator that removes duplicates in a multiset globally and compares whole multiset elements. For the vector operator `distinctV`, however, we have to observe the segment structure and eliminate duplicates only relative to one particular segment. Furthermore, only the payload components of the multiset elements are compared — indexes and order labels are to be ignored.

$$\mathcal{R}[\![\texttt{distinctV}\ V]\!]\rho = q_d \qquad (\mathcal{MA}\text{-DISTINCT})$$
$$\text{where}\quad q_n \equiv \#\{\lambda x.\langle \texttt{x.s}, \texttt{x.p}\rangle, \lambda x.\texttt{x.o}\}\ \rho(V)$$
$$q_s \equiv \sigma\{\lambda x.\texttt{x.2 = 1}\}\ q_n$$
$$q_d \equiv \pi\{\lambda x.\texttt{x.1}\}\ q_s$$

Based on #{}, rule ($\mathcal{MA}$-DISTINCT) partitions the input such that partition elements concur in their outer indexes and payload values. A segment-relative elimination of duplicates can then be performed by selecting exactly one element — the first — from each partition. Note that partition elements are enumerated in their relative order indicated by their order labels. This allows to preserve the semantics of the vector operator `distinctV` that keeps the first unique element (Section 4.3.2.2).

Next to ($\mathcal{MA}$-NUMBER), ($\mathcal{MA}$-DISTINCT) is the only rule where row numbering is necessary to faithfully implement the operator semantics. Selecting the *first* unique element is not possible when eliminating duplicates with the δ operator: order labels must not be part of the δ input nor can they be recovered afterwards[2].

2 PostgreSQL's **`DISTINCT ON(...)`** construct provides for a more direct implementation of `distinctV` without #{}.

FOLDING AND GROUPING As discussed in Section 6.2.2, folding of segments is implemented using `Γ{}` by grouping on the outer index ($\mathcal{MA}$-REDUCE).

$$\mathcal{R}[\![\texttt{reduceV\{}s_z\texttt{,}\ s_f\texttt{\}}\ V]\!]_\rho = q_d \qquad (\mathcal{MA}\text{-REDUCE})$$
$$\text{where}\quad q_g \equiv \Gamma\{\lambda x.x.s, s_z, \lambda x\,y.s_f\ x.p\ y\}\ \rho(V)$$
$$q_d \equiv \pi\{\lambda x.\langle\!\langle x.1, x.1, \langle\rangle, x.2\rangle\!\rangle\}\ q_g$$

Note that we have to direct the folding function to the payload of the input only. In the result, all segments have exactly one element. Using $\langle\rangle$ as order label is sufficient.

Rule ($\mathcal{MA}$-GROUP) implements unary grouping without aggregation.

$$\mathcal{R}[\![\texttt{groupV\{f\}}\ V]\!]_\rho = (q_o, q_i, q_m) \qquad (\mathcal{MA}\text{-GROUP})$$
$$\text{where}\quad q_1 \equiv \pi\{\lambda x.\langle x, f\ x.p\rangle\}\ \rho(V)$$
$$q_2 \equiv \pi\{\lambda x.\langle x.1.s, x.2\rangle\}\ q_1$$
$$q_3 \equiv \delta\ q_2$$
$$q_o \equiv \pi\{\lambda x.\langle\!\langle x.1, x, x.2, x.2\rangle\!\rangle\}\ q_3$$
$$q_i \equiv \pi\{\lambda x.\langle\!\langle\langle x.1.s, x.2\rangle, x.1.k, x.1.o, x.1.p\rangle\!\rangle\}\ q_1$$
$$q_m \equiv \pi\{\lambda x.x.k\}\ q_s$$

The plan q_1 pairs each element of the input with the corresponding grouping key `f x.p`. Plan q_1 serves as the basis for the outer as well as the inner vector. Operator `groupV{}` refines each segment individually by splitting it into multiple segments for each unique value of the grouping key. Every unique combination of outer index and grouping key describes one segment in the result.

These unique combinations are computed in q_3 and used in q_o to specify element identity (inner index) for the outer vector q_o. Grouping keys form the payload of the outer vector, but also order labels. This implements the first aspect of the order semantics of vector operator `groupV{}`: groups themselves are ordered by the grouping key. Naturally, there is no guarantee for grouping keys to be unique over the complete outer vector q_o. As we require only per-segment order, this is not necessary.

The multiset q_i that encodes the inner vector is computed by a single projection. It has the same shape as the original input q — we only have to redistribute the elements to the new, more fine-grained segments. Redistribution is easily achieved by combining the original inner index with the grouping key. Inner indexes stay valid. Note that we reuse the order labels from q. The original order labels are completely sufficient to describe the *relative* order of elements of the new segments. This implements the second aspect of the order semantics of `groupV{}`: elements in the individual groups appear in the order of the original input. As in ($\mathcal{MA}$-SORT), the ordering transformation q_m just serves to preserve the structure of the vector program and will never actually be used.

The combination of outer index and grouping key link the outer and inner vectors q_o and q_i. Natural indexes enable our implementation in which the only actual work is the δ operator for the outer vector. The effort shared between the outer and inner vector consists just of the evaluation of the scalar expression `f` in q_1. An implementation based on synthetic indexes, on the other hand, would generate the linking indexes by computing integer ranks with a window function based on the result of `f`. Used as indexes, the ranks would be shared between the outer and inner index and have to be

kept consistent. Just as for the example in Figure 59b, we would have vectors linked by the result of an expensive operator that is hard to remove.

Operator `groupaggV{}` refines segments based on the grouping key as well, but folds the resulting segments immediately.

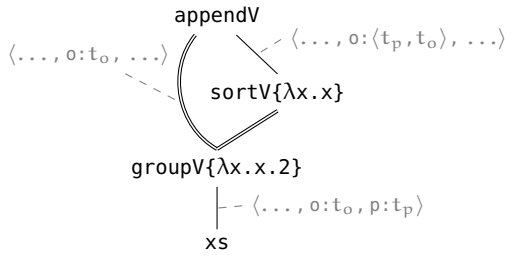
$$\mathcal{R}[\![\texttt{groupaggV\{}s_g\texttt{, }s_z\texttt{, }s_f\texttt{\} V}]\!]\rho = q_d \qquad (\mathcal{MA}\text{-GROUPAGG})$$
$$\text{where} \quad q_g \equiv \Gamma\{\lambda x.\langle \texttt{x.s}, s_f\ \texttt{x}\rangle, s_z, \lambda x\,\texttt{y.}s_g\ \texttt{x.p y}\}\ \rho(V)$$
$$q_d \equiv \pi\{\lambda x.\langle\!\langle \texttt{x.1.1, x.1, x.1.2, }\langle \texttt{x.1.2, x.2}\rangle\rangle\!\rangle\}\ q_g$$

In rule ($\mathcal{MA}$-GROUPAGG), `groupaggV{}` maps to its $\mathcal{MA}$ counterpart $\Gamma\{\}$. The input is grouped by outer index and grouping key and folded according to s_z and f.

APPENDING SEGMENTS Natural indexes and lazy order avoid numbering operators as far as possible. So far, we have employed numbering operators only to implement `numberV` and `distinctV`, but not for index and order maintenance. Unfortunately, `appendV` is the sole exception.

Consider the following $\mathcal{CL}$ query: Each group produced by the `group` combinator is append with itself after sorting.

```
[ append g.2
         (sort [⟨x,x⟩
                | x ← g.2 ])
| g ← group xs ]
```



The corresponding vector plan on the right side is annotated with the element types of the multisets that encode the vectors. The type of order labels in the left input of `appendV` is the same as in the input of `groupV{}`. The right input of `appendV`, though, has a different type of order labels: `sortV{}` pairs the payload type with the original label type to implement sorting.

Non-uniform index and order label types pose a problem for the implementation of `appendV`: We can't simply employ multiset union $\cup$ since the element types of the input do not match. Instead, we have to provide uniform indexes and order labels.

The helper function $\text{uniform}_s(-)$ uses `#{}` to provide uniform `Int` indexes and order labels. An enumeration in the order of the unique combination of outer index and order label generates unique indexes. Note that the generated values do not represent segment positions but still encode the relative order of segment elements. $\text{uniform}_s(-)$ prefixes indexes and order labels with a constant scalar value s. As we change the inner index of a vector, we have to maintain consistency with any inner vectors. The rekeying transformation q_m updates outer index values in inner vectors to conform to the new synthetic index.

$$\text{uniform}_s(q) = (q_d, q_m)$$
$$\text{where} \quad q_n \equiv \texttt{\#\{}\lambda x.\langle\rangle, \lambda x.\langle \texttt{x.s, x.o}\rangle\texttt{\}}\ q$$
$$q_s \equiv \pi\{\lambda x.\langle \texttt{x.1, }\langle s, \texttt{x.2}\rangle\rangle\}\ q_n$$
$$q_m \equiv \pi\{\lambda x.\langle \texttt{f=x.1.k, t=x.2}\rangle\}\ q_s$$
$$q_d \equiv \pi\{\lambda x.\langle\!\langle \texttt{x.1.s, x.2, x.2, x.1.p}\rangle\!\rangle\}\ q_s$$

With uniform vectors produced by $\text{uniform}_s(-)$, implementing `appendV` comes down to using $\cup$ in rule ($\mathcal{MA}$-APPEND). In q_{u_1} and q_{u_2}, indexes and order labels overlap. To ensure that indexes in the result of $\cup$ are unique and that order labels encode the correct segment order (elements from the left before elements from the right), $\text{uniform}_s(-)$ prefixes indexes and order labels with integer tags 1 and 2.

$$\begin{aligned}
&\mathcal{R}[\![\texttt{appendV}\ V_1\ V_2]\!]\rho = (q_d, q_m, q'_m) && (\mathcal{MA}\text{-APPEND})\\
&\quad\text{where}\quad (q_{u_1}, q_{m_1}) \equiv \text{uniform}_1(\rho(V_1))\\
&\qquad\qquad (q_{u_2}, q_{m_2}) \equiv \text{uniform}_2(\rho(V_2))\\
&\qquad\qquad\quad q_d \equiv q_{u_1} \cup q_{u_2}
\end{aligned}$$

JOIN OPERATORS The implementation of vector join operators has been outlined in Section 6.2.

$$\begin{aligned}
&\mathcal{R}[\![\texttt{thetajoinV\{}s_p\texttt{\}}\ V_1\ V_2]\!]\rho = (q_d, q_m, q'_m) && (\mathcal{MA}\text{-THETAJOIN})\\
&\quad\text{where}\quad q_j \equiv \rho(V_1) \bowtie\texttt{\{}\lambda x\,y.x.s = y.s \wedge s_p\ x.p\ y.p\texttt{\}}\ \rho(V_2)\\
&\qquad\qquad q_d \equiv \pi\texttt{\{}\lambda x.\langle\!\langle \texttt{x.1.s},\ \text{pair}_k(x),\ \text{pair}_o(x),\ \text{pair}_p(x)\rangle\!\rangle\texttt{\}}\ q_j\\
&\qquad\qquad q_m \equiv \pi\texttt{\{}\lambda x.\langle \texttt{f = x.1.k},\ \texttt{t} = \langle \texttt{x.1.k}, \texttt{x.2.k}\rangle\rangle\texttt{\}}\ q_j\\
&\qquad\qquad q'_m \equiv \pi\texttt{\{}\lambda x.\langle \texttt{f = x.2.k},\ \texttt{t} = \langle \texttt{x.1.k}, \texttt{x.2.k}\rangle\rangle\texttt{\}}\ q_j
\end{aligned}$$

Rule ($\mathcal{MA}$-THETAJOIN) is centered around a $\bowtie\texttt{\{\}}$ operator that performs the actual work. The projection q_d pairs (1) inner indexes to provide unique indexes, (2) order labels to implement list join semantics, and (3) payloads. Projections q_m and q'_m generate replication transformations that maintain consistency with any inner vectors.

Even less effort is necessary for the `semijoinV{}` operator. In addition to its $\mathcal{MA}$ equivalent $\ltimes\texttt{\{\}}$, we only have to provide a map that eliminates stale segments from any inner vectors.

$$\begin{aligned}
&\mathcal{R}[\![\texttt{semijoinV\{}s_p\texttt{\}}\ V_1\ V_2]\!]\rho = (q_d, q_m) && (\mathcal{MA}\text{-SEMIJOIN})\\
&\quad\text{where}\quad q_j \equiv \rho(V_1) \ltimes\texttt{\{}\lambda x\,y.x.s = y.s \wedge s_p\ x.p\ y.p\texttt{\}}\ \rho(V_2)\\
&\qquad\qquad q_m \equiv \pi\texttt{\{}\lambda x.x.k\texttt{\}}\ q_j
\end{aligned}$$

Rule $\mathcal{MA}$-ANTIJOIN for operator `antijoinV{}` is exactly equivalent to $\mathcal{MA}$-SEMIJOIN with $\triangleright\texttt{\{\}}$ being used instead of $\ltimes\texttt{\{\}}$.

$$\begin{aligned}
&\mathcal{R}[\![\texttt{antijoinV\{}s_p\texttt{\}}\ V_1\ V_2]\!]\rho = (q_d, q_m) && (\mathcal{MA}\text{-ANTIJOIN})\\
&\quad\text{where}\quad q_j \equiv \rho(V_1) \triangleright\texttt{\{}\lambda x\,y.x.s = y.s \wedge s_p\ x.p\ y.p\texttt{\}}\ \rho(V_2)\\
&\qquad\qquad q_m \equiv \pi\texttt{\{}\lambda x.x.k\texttt{\}}\ q_j
\end{aligned}$$

As outlined before, operator `nestjoinV{}` maps to a regular $\bowtie\texttt{\{\}}$.

$$\begin{aligned}
&\mathcal{R}[\![\texttt{nestjoinV\{}s_p\texttt{\}}\ V_1\ V_2]\!]\rho = (q_d, q_m, q'_m) && (\mathcal{MA}\text{-NESTJOIN})\\
&\quad\text{where}\quad q_j \equiv \rho(V_1) \bowtie\texttt{\{}\lambda x\,y.x.s = y.s \wedge s_p\ x.p\ y.p\texttt{\}}\ \rho(V_2)\\
&\qquad\qquad q_d \equiv \pi\texttt{\{}\lambda x.\langle\!\langle \texttt{x.1.k},\ \text{pair}_k(x),\ \texttt{x.2.o},\ \text{pair}_p(x)\rangle\!\rangle\texttt{\}}\ q_j\\
&\qquad\qquad q_m \equiv \pi\texttt{\{}\lambda x.\langle \texttt{f = x.1.k},\ \texttt{t} = \text{pair}_k(x)\rangle\texttt{\}}\ q_j\\
&\qquad\qquad q'_m \equiv \pi\texttt{\{}\lambda x.\langle \texttt{f = x.2.k},\ \texttt{t} = \text{pair}_k(x)\rangle\texttt{\}}\ q_j
\end{aligned}$$

Whereas `thetajoinV{}` preserves the segment structure of the operands, ($\mathcal{MA}$-NESTJOIN) uses the left operands' inner index to create new segments.

This provides us with the opportunity to minimize order constraints. Operator `thetajoinV{}` preserves the outer index of elements from the left and right operand and pairs their order labels to describe the relative order correctly. As `nestjoinV{}` creates a new segment for every element of the left operand, order labels from the left will be constant in each new segment. Recall that we only require order labels to describe the order in an individual segment. Order labels of the right operand are perfectly sufficient to describe the relative order of elements in the segments of the result vector.

The last remaining $\mathcal{SL}$ join operator, `groupjoinV{}`, again maps directly to its $\mathcal{MA}$ equivalent.

$$\mathcal{R}[\![\texttt{groupjoinV\{}s_p\texttt{,}\ s_z\texttt{,}\ s_f\texttt{\}}\ V_1\ V_2]\!]_\rho = q_d \qquad \text{(MA-GROUPJOIN-2)}$$

where

$$q_j \equiv \rho(V_1) \rtimes\!\bowtie \{\lambda \texttt{x y.} \wedge\ \texttt{x.s = y.s}\ s_p\ \texttt{x.p y.p,}\ s_z\texttt{,}\ \lambda \texttt{a V.}\ s_f\ \texttt{a}\ \text{pair}_p(V)\}\rho(V_2)$$

$$q_d \equiv \pi\{\lambda \texttt{x.}\langle\!\langle \texttt{x.1.s, x.1.k, x.1.o,}\ \langle \texttt{x.1.p, x.2}\rangle\rangle\!\rangle\}\ q_j$$

Note that our usage of $\rtimes\!\bowtie\{\}$ relies on the multiset $\rho(V_1)$ not containing duplicates. By definition, a multiset encoding of a segment vector includes element identity in the form of index values and thus is free of duplicates.

In Chapter 5 we have employed list-based join combinators and the equivalent order-preserving segment join operators as crucial tools to specialize (nested) iteration patterns. All $\mathcal{SL}$ join operators lower to a single $\mathcal{MA}$ join combined with a projection. Effectively, the cost of a vector join is that of the underlying $\mathcal{MA}$ join. This supports our claim that we can indeed derive *idiomatic* and *efficient* relational queries through *Query Flattening*.

ADMINISTRATIVE OPERATORS To conclude the description of the $\mathcal{MA}$ code generator, we list translation rules for operators that maintain administrative information on multiset vectors.

$$\mathcal{R}[\![\texttt{segmentV}\ V]\!]_\rho = \pi\{\lambda \texttt{x.}\langle\!\langle \texttt{x.k, x.k, x.o, x.p}\rangle\rangle\!\rangle\}\ \rho(V) \qquad (\mathcal{MA}\text{-SEGMENT})$$

$$\mathcal{R}[\![\texttt{unsegmentV}\ V]\!]_\rho = \pi\{\lambda \texttt{x.}\langle\!\langle \langle\rangle\texttt{, x.k, x.o, x.p}\rangle\rangle\!\rangle\}\ \rho(V) \qquad (\mathcal{MA}\text{-UNSEGMENT})$$

$$\mathcal{R}[\![\texttt{mergesegV}\ V_o\ V_i]\!]_\rho = q_d \qquad (\mathcal{MA}\text{-MERGEMAP})$$

where

$$q_j \equiv \rho(V_1) \ltimes\{\lambda \texttt{x y.x.k = y.s}\}\ \rho(V_2)$$

$$q_d \equiv \pi\{\lambda \texttt{x.}\langle\!\langle \texttt{x.1.s, x.2.k,}\ \text{pair}_o(\texttt{x})\texttt{, x.2.p}\rangle\rangle\!\rangle\}\ q$$

$$\mathcal{R}[\![\texttt{alignV}\ V_1\ V_2]\!]_\rho = q_d \qquad (\mathcal{MA}\text{-ALIGN})$$

where

$$q_j \equiv \rho(V_1) \ltimes\{\lambda \texttt{x y.x.k = y.k}\}\ \rho(V_2)$$

$$q_d \equiv \pi\{\lambda \texttt{x.}\langle\!\langle \texttt{x.1.s, x.1.k, x.1.o,}\ \text{pair}_p(\texttt{x})\rangle\rangle\!\rangle\}\ q$$

The implementation of `unboxV{s}` uses a left outerjoin to account for missing segments.

$$\mathcal{R}[\![\texttt{unboxV\{}s_z\texttt{\}}\ V_1\ V_2]\!]_\rho = q_d \qquad \text{(MA-UNBOX-2)}$$

where

$$q_j \equiv \rho(V_2) \rtimes\!\bowtie \{\lambda \texttt{x y.x.k = y.s,}\ s_z\texttt{,}\ \lambda \texttt{x.x.p}\}\ \rho(V_2)$$

$$q_d \equiv \pi\{\lambda \texttt{x.}\langle\!\langle \texttt{x.1.s, x.1.k, x.1.o,}\ \langle \texttt{x.1.p, x.2}\rangle\rangle\!\rangle\}\ q_j$$

($\mathcal{MA}$-REP) uses the multiset product $\times$ to replicate the unit segment of the left operand for all elements of the right operand.

$$\mathcal{R}[\![\texttt{repV}\ V_1\ V_2]\!]_\rho = (q_d, q_m) \qquad (\mathcal{MA}\text{-REP})$$
$$\text{where}\quad q_p \equiv \rho(V_1) \times \rho(V_2)$$
$$q_d \equiv \pi\{\lambda x.\langle\!\langle \texttt{x.2.k},\ \text{pair}_k(\texttt{x}), \texttt{x.1.o}, \texttt{x.1.p}\rangle\!\rangle\}\ q_p$$
$$q_m \equiv \pi\{\lambda x.\langle \texttt{f}=\texttt{x.2.k}, \texttt{t}=\langle \texttt{x.2.k}, \texttt{x.1.k}\rangle\rangle\}\ q_p$$

Similar to ($\mathcal{MA}$-NESTJOIN), we can ignore the order labels from the right operand because `repV` creates new segments.

$$\mathcal{R}[\![\texttt{repsegV}\ V_1\ V_2]\!]_\rho = (q_d, q_m) \qquad (\mathcal{MA}\text{-REPSEG})$$
$$\text{where}\quad q_j \equiv \rho(V_1) \ltimes\{\lambda x\, y.\texttt{x.k}=\texttt{y.s}\}\ \rho(V_2)$$
$$q_d \equiv \pi\{\lambda x.\langle\!\langle \texttt{x.2.s}, \texttt{x.2.k}, \texttt{x.2.o}, \texttt{x.1.p}\rangle\!\rangle\}\ q_j$$
$$q_m \equiv \pi\{\lambda x.\langle \texttt{f}=\texttt{x.1.k}, \texttt{t}=\texttt{x.2.k}\rangle\}\ q_p$$

$$\mathcal{R}[\![\texttt{combineV}\ V_b\ V_1\ V_2]\!]_\rho = q_d \qquad (\mathcal{MA}\text{-COMBINE})$$
$$\text{where}\quad q_u \equiv \cup\, \rho(V_1)\, \rho(V_2)$$
$$q_j \equiv \rho(V_b) \ltimes\{\lambda x\, y.\texttt{x.k}=\texttt{y.s}\}\ q_u$$
$$q_d \equiv \pi\{\lambda x.\langle\!\langle \texttt{x.1.s}, \texttt{x.1.k}, \texttt{x.1.o}, \texttt{x.2.p}\rangle\!\rangle\}\ q_j$$

INDEX PROPAGATION Finally, we implement the four $\mathcal{SL}$ operators that propagate index changes from outer to inner vectors. Here, we benefit from having distinguished different forms of index propagation (Section 4.3.2). We list the operator implementations in decreasing order of the evaluation work involved.

In rules ($\mathcal{MA}$-APPREP) and ($\mathcal{MA}$-APPKEY), operator $\ltimes\{\}$ joins a vector with a segment map to update outer indexes. `apprepV` applies a replication map that describes a replication of segments. In the multiset implementation, each element of multiset $\rho(V)$ may find multiple or none join partners. In effect, segments are replicated or removed according to the replication map.

$$\mathcal{R}[\![\texttt{apprepV}\ I\ V]\!]_\rho = (q_d, q_m) \qquad (\mathcal{MA}\text{-APPREP})$$
$$\text{where}\quad q_j \equiv \rho(I) \ltimes\{\lambda x\, y.\texttt{x.f}=\texttt{y.s}\}\ \rho(V)$$
$$q_d \equiv \pi\{\lambda x.\langle\!\langle \texttt{x.1.t}, \langle \texttt{x.2.k}, \texttt{x.1.t}\rangle, \texttt{x.2.o}, \texttt{x.2.p}\rangle\!\rangle\}\ q_j$$
$$q_m \equiv \pi\{\lambda x.\langle \texttt{f}=\texttt{x.2.k}, \texttt{t}=\langle \texttt{x.2.k}, \texttt{x.1.t}\rangle\rangle\}\ q_p$$

A rekeying transformation applied by `appkeyV`, on the other hand, describes a one-to-one mapping between outer indexes. In rule ($\mathcal{MA}$-APPKEY), every element of the right join operand is guaranteed to find exactly one join partner. Consequently, the cardinality of multisets $\rho(V)$ and q_d is the same.

$$\mathcal{R}[\![\texttt{appkeyV}\ I\ V]\!]_\rho = q_d \qquad (\mathcal{MA}\text{-APPKEY})$$
$$\text{where}\quad q_j \equiv \rho(I) \ltimes\{\lambda x\, y.\texttt{x.f}=\texttt{y.s}\}\ \rho(V)$$
$$q_d \equiv \pi\{\lambda x.\langle\!\langle \texttt{x.1.t}, \texttt{x.2.k}, \texttt{x.2.o}, \texttt{x.2.p}\rangle\!\rangle\}\ q_j$$

In rule ($\mathcal{MA}$-APPFILTER), a semijoin is sufficient to filter segments according to a segment filter.

$$\mathcal{R}[\![\texttt{appfilterV}\ I\ V]\!]_\rho = (q_d, q_m) \qquad (\mathcal{MA}\text{-APPFILTER})$$
$$\text{where}\quad q_j \equiv \rho(V) \ltimes\{\lambda x\, y.\texttt{x.s}=\texttt{y}\}\ \rho(I)$$
$$q_m \equiv \pi\{\lambda x.\texttt{x.k}\}\ q_j$$

Finally, rule ($\mathcal{MA}$-APPSORT) does not invoke any runtime effort. As we track the order of vector elements relative to their segments only, the reordering of segments described by a sort transformation has no meaning in the multiset code generator. The implementation of `appsortV` ignores the sort transformation and returns the vector unmodified. We produce a further sort transformation q_m to conform with the type of the $\mathcal{SL}$ operator `appsortV`.

$$\mathcal{R}[\![\texttt{appsortV I V}]\!]_\rho = (\rho(V), q_m) \qquad (\mathcal{MA}\text{-APPSORT})$$
$$\text{where} \quad q_m \equiv \pi\{\lambda\texttt{x.x.k}\}\ \rho(V)$$

6.3.1 *Flattening and Relational Query Optimization*

In Chapter 5, we use `nestjoinV{}` as an essential tool for the optimization of nested queries. Steenhagen *et al.* [SAB94] note that nestjoin does not have the algebraic properties of regular join operators. In particular, it is neither associative nor commutative and does not associate with a regular theta join. The corresponding vector operator `nestjoinV{}` is defined on flat collections but suffers from the same lack of algebraic properties. Here, the necessity to preserve the segment structure prohibits associativity and commutativity.

However, relational query evaluation critically relies on join ordering based on these properties to limit the size of intermediate results [Lei+15]. Due to the lack of algebraic properties, a direct physical implementation of either the $\mathcal{CL}$ or $\mathcal{SL}$ nestjoin would severely limit an optimizer's freedom to re-order join trees. The $\mathcal{MA}$ implementation of `nestjoinV{}`, on the other hand, enjoys the crucial algebraic properties and integrates well with query optimization in the relational backend system.

Vector operator implementation by $\mathcal{MA}$ operators does not prescribe a particular evaluation strategy on the backend database system. As we target a logical algebra, the backend is free to choose among the evaluation strategies implemented by the backends' query engine: $\ltimes\{\}$, for example, can be evaluated with sorting, hashing or based on an index.

One cause of concern could be that adding segment identifier comparisons to join predicates, grouping specifications and the like might prevent index usage. However, this will turn out not to be a problem. In those cases in which evaluation actually can be backed by an index, we are typically able to infer statically that outer indexes are constant. In these cases, index comparisons can be eliminated from the backend plan. We will elaborate on this in Section 6.4.

6.3.2 *Running Example*

In Figure 67 we show the $\mathcal{MA}$ plan for our running example Query Q12. This plan is the result of lowering the $\mathcal{SL}$ plan of Figure 52 to $\mathcal{MA}$. We focus on the structure of the plan and omit operator arguments. This plan is the direct output of the $\mathcal{MA}$ translation scheme described in this chapter. No $\mathcal{MA}$ optimizations have been applied (see Section 6.4).

The structure of the plan directly reflects the structure of the original $\mathcal{SL}$ plan. Each $\mathcal{SL}$ operator has been expanded into one or few $\mathcal{MA}$ operators as dictated by the translation rules. At the plan bottom, both base tables are replicated over the *singleton* literal multiset that encodes the list $[\langle\rangle]$. This is the price we pay for uniformly employing lifted combinators in query flat-

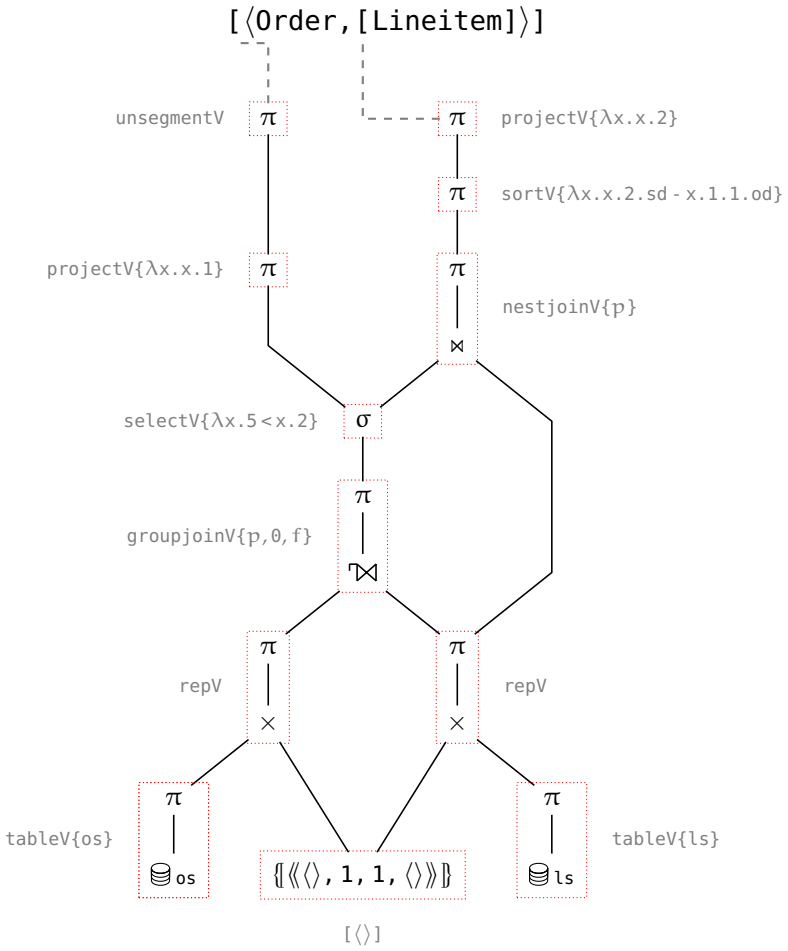


Figure 67: Structure of the $\mathcal{MA}$ plan implementing the running example Query Q12. Obtained by lowering the $\mathcal{SL}$ program of Figure 52. No $\mathcal{MA}$ optimizations have been applied.

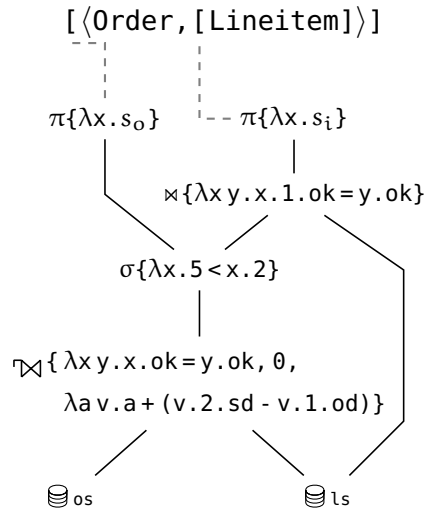


Figure 68: $\mathcal{MA}$ plan for running example Query Q12 from unoptimized plan in Figure 67.

tening (Section 4.1). A cartesian product with a singleton input is unlikely to reflect in the runtime of the relational query at all. Still, this pattern can be completely removed by rewriting the $\mathcal{MA}$ plan.

In the $\mathcal{MA}$ plan, we can expect operators ⟕{}, σ{} and ⋈{} to involve non-trivial evaluation effort. These operators directly express the core query logic of the original $\mathcal{CL}$ query and can not be avoided. Note that segment sorting of the inner vector maps to a projection — any runtime effort for actual sorting is delayed.

The plan faithfully implements the list semantics of the original $\mathcal{CL}$ query. It maintains element order as well as index information to describe nested data. Still, this effort barely reflects in the plan. In particular, it does not feature any numbering operators. This is to be expected since the original query does not involve `numberV`, `appendV` and `distinctV` — the only $\mathcal{SL}$ operators whose lowering emits a numbering operator. Maintaining administrative information in our plan is purely expressed in projections.

6.4 OPTIMIZATION OF RELATIONAL PLANS

The $\mathcal{MA}$ plan for our running example in Figure 67 is arguably reasonable. However, it still contains potential for simplification. In Figure 68 we show the $\mathcal{MA}$ plan after a number of simple rewrites have been applied. The following rewrites have been performed.

- Cartesian products with the singleton literal multiset at the plan bottom have been replaced with a projection.
- These projections and all other projections have been pushed downstream and inlined with other projections.
- As a result of projection inlining, the part of the join predicates that compares outer index values (segment identifiers) has been recognized as being constant `True` and eliminated.

In the rewritten plan, the only remaining projections are at the plan top. They directly describe the multiset encoding of the vectors that result from

evaluating our query of type `[⟨Order,[Lineitem]⟩]`. In the plan, expression s_o generates the multiset encoding of the outer vector. As it encodes a top-level list, the outer index is $\langle\rangle$. Element identity is described by inner indexes derived from `o_orderkey` (abbreviated as `os`), the primary key of the `orders` base table. The same primary key also defines the element order in the outer list. As dictated by the element type of the outer list, the payload consists of pairs of order records and the nested list placeholder $\langle\rangle$.

$$
\begin{aligned}
s_o = \langle\, & \mathtt{s} = \langle\rangle, \\
& \mathtt{k} = \langle\langle\rangle, \mathtt{x.1.ok}\rangle, \\
& \mathtt{o} = \mathtt{x.1.ok}, \\
& \mathtt{p} = \langle \mathtt{x.1}, \langle\rangle\rangle\rangle
\end{aligned}
$$

Expression s_i generates the encoding of the inner vector. Crucially, we note that the outer index generated here matches the inner index generated by s_o — in the inner vector, segments are identified by `o_orderkey`. Elements of the inner vector, in turn, are identified by inner indexes composed from the outer index and the primary key of the `lineitem` base table (`l_orderkey` and `l_linenumber` abbreviated as `ok` and `ln`, respectively). Together, these attributes uniquely identify elements in the join result of `orders` and `lineitem`. Order labels in the inner vector reflect the original `sort` combinator. The order of segment elements in the inner vector is dictated primarily by the difference between `l_shipdate` and `o_orderdate` (`sd`, `od`) according to the original query logic.

$$
\begin{aligned}
s_i = \langle\, & \mathtt{s} = \langle\langle\rangle, \mathtt{x.1.ok}\rangle, \\
& \mathtt{k} = \langle\langle\langle\rangle, \mathtt{x.1.ok}\rangle, \langle\rangle, \langle \mathtt{x.2.ok}, \mathtt{x.2.ln}\rangle\rangle, \\
& \mathtt{o} = \langle \mathtt{x.2.sd - x.1.1.od}, \langle \mathtt{x.2.ok}, \mathtt{x.2.ln}\rangle\rangle, \\
& \mathtt{p} = \mathtt{x.2}\rangle
\end{aligned}
$$

In this work, we do not further delve into relational query optimization. Logical optimization of relational algebra plans is extensively discussed in the literature [JK84]. Prior work on optimization can be directly transferred to $\mathcal{MA}$. For example, the set of relational optimizations described by Rittinger [Rit11] can mostly be transferred to $\mathcal{MA}$. Rittinger's rewrites for the simplification and elimination of numbering operators can be employed in those cases where our translations scheme actually introduces numbering.

It is important to note, though, that any optimizations that are specific to the nature of $\mathcal{CL}$ (*i.e.* nested and ordered lists, nested iteration) as well as *Query Flattening* are performed at higher levels of abstraction. $\mathcal{MA}$ plans produced from $\mathcal{CL}$ via query flattening do not require large-scale structural changes for query unnesting and similar optimizations. Typically, these $\mathcal{MA}$ plans only provide opportunities for simple house-cleaning rewrites as in our running example.

6.5 SQL CODE GENERATION

One step is missing to translate flattened queries to SQL. The lowering to $\mathcal{MA}$ uses nested records. In the translation rules, the support for nested records in $\mathcal{MA}$ has proven convenient: We can focus on the core aspects of indexes and order labels without having to keep track of attribute names. Nested records allow for an easy composition of indexes and order labels by forming pairs. The tracking of column names and renaming of columns

necessary for record flattening would considerably inflate the translation rules.

To real-world relational systems, nested records are not alien: *Object-relational* and *NewSQL* systems support structured records in relations, for example in the form of user-defined *composite types* (PostgreSQL [PG]) or *Protocol Buffers* (Google F1 [Shu+12]). These are non-standard vendor-specific extensions, though. At the same time, relevant relational systems (*e.g.* HyPer, MonetDB) do not support structured values at all.

Nested records can be simulated with flat records that contain only the base values at the leaf of the nested records. Information about the record nesting structure can be encoded in the record labels. To flatten records, it is sufficient to modify the arguments of $\mathcal{MA}$ operators. Crucially, this does not change the structure of $\mathcal{MA}$ plans. In this thesis, we do not expand on record flattening. The translation is described by Cheney *et al.* [CLW14b].

After record flattening, it is straightforward to generate actual SQL code from relational plans. A SQL code generator that uses *tiling* to generate compact SQL:2003 queries from relational algebra plans has been described by [May13]. Our implementation of *Query Flattening* in DSH is backed by a similar code generator.

6.6 RELATED WORK

The *Loop-Lifting* translation (Section 2.2.2) heavily relies on numbering operators to derive compact integer index values and order labels. Indexes are derived by a row-numbering operator implemented via the SQL `row_number()` window function. Order labels are computed by a ranking operator implemented with `dense_rank()`. The resulting relational plans are littered with numbering operators that require sorting effort in the relational backend and restrict the backends freedom to optimize plans.

In contrast, we pick up suggestions by Rittinger [Rit11, Section 6.3.2] as well as Cheney *et al.* [CLW14a] and implement *natural* indexes and order labels that track the columns that express element identity and order. We can precisely characterize which plans involve numbering operators. Any $\mathcal{CL}$ query that does not append lists, eliminate duplicates or actually enumerates elements will not feature numbering operators — without relying on an optimization step.

Rittinger [Rit11, Section 4.3] describes an involved set of algebraic rewrites that simplify and eliminate numbering operators in *Loop-Lifting* plans based on global plan properties. Under certain circumstances, numbering operators are promoted to less expensive variants. Ranking operators used for order labels can be eliminated if their ordering criterion is a single column and only the relative order of rank values is required. The effect of removing rank operators mimics natural order labels in some special cases but is hindered by the *Loop-Lifting* translation scheme: *Loop-Lifting* rules hardcode a *single attribute* that expresses element order. Hence, any scenario in which the order of elements is dictated by multiple attributes prohibits removal of rank operators.

7 DELAYED REPLICATION

By default, *Query Flattening* implements nested iteration by replicating base tables. We have described the resulting problematic patterns in flattened queries in Section 4.3.4. In Chapter 5 we show that replication can be eliminated by rewriting $\mathcal{CL}$ expressions prior to query flattening. Correlated nested iteration is encapsulated in join combinators and loop-invariant expressions are lifted out of iterators. These optimizations eliminate replication in many relevant cases.

Certain occurrences of replication, however, can not be eliminated by rewriting $\mathcal{CL}$ queries. As an example, we pick up Query Q13 from Chapter 5. Recall that the universal quantifier in the original query correlates `zs` with both `xs` and `ys`. This prevents us from lifting the `antijoin{}` combinator out of the head of the outer comprehension.

```
[ [ y.2 | y ← antijoin{p2} xy.2 zs ]
| xy ← nestjoin{p1} xs ys ]
```

The `antijoin{}` combinator is applied iteratively to each element in the result of `nestjoin{}` and joins it with `zs`. Lifting maps explicit iteration to the lifted combinator `antijoin{}`$^{\uparrow}$. To enable a uniform data-parallel evaluation of `antijoin{}`$^{\uparrow}$, lifting replicates `zs` (Rule LIFT-TABLE) and provides an independent copy of `zs` for each iteration. The problematic replication is clearly visible in the resulting $\mathcal{SL}$ plan in Figure 69. The segment operator `antijoinV{}` requires its operands to have the same segment structure. The plan creates a vector that matches the segment structure of the left `antijoinV{}` operand by replicating `tableV{zs}`. A relational plan derived from this $\mathcal{SL}$ plan will effectively compute the cartesian product of `zs` and `xs`. This is clearly not acceptable.

We incur replication in this manner whenever base table references can't be lifted out of comprehensions — for example due to correlation across multiple iteration levels. The same problem can be observed in queries that construct deeply nested lists from nested comprehensions as in the

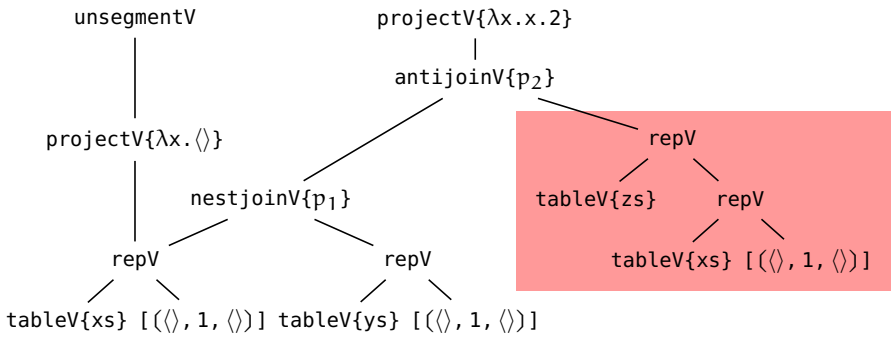


Figure 69: Query Q13 shreds into this $\mathcal{SL}$ plan. The plan region marked red effectively computes the cartesian product of `xs` and `zs`.

following example. Note that the inner predicate p_2 correlates zs with both xs and ys.

$$
\begin{aligned}
&[\ \langle \mathtt{x}, [\ \langle \mathtt{y}, [\ \mathtt{z} \mid \mathtt{z} \leftarrow \mathtt{zs}, p_2\ \mathtt{x}\ \mathtt{y}\ \mathtt{z}\ ]\rangle \mid \mathtt{y} \leftarrow \mathtt{ys}, p_1\ \mathtt{x}\ \mathtt{y}\ ]\rangle \\
&\mid \mathtt{x} \leftarrow \mathtt{xs}\ ] \\
&\equiv \quad \{\ \textsc{nestjoin-head}\ \} \\
&\quad [\ \langle \mathtt{x.1}, [\ \langle \mathtt{y.2}, [\ \mathtt{z} \mid \mathtt{z} \leftarrow \mathtt{zs}, p_2\ \mathtt{y.1}\ \mathtt{y.2}\ \mathtt{z}\ ]\rangle \mid \mathtt{y} \leftarrow \mathtt{x.2}\ ]\rangle \\
&\quad \mid \mathtt{x} \leftarrow \mathtt{nestjoin}\{p_1\}\ \mathtt{xs}\ \mathtt{ys}\ ] \\
&\equiv \quad \{\ \textsc{nestjoin-head}\ \} \\
&\quad [\ \langle \mathtt{x.1}, [\ \langle \mathtt{y.1.2}, [\ \mathtt{z.2} \mid \mathtt{z} \leftarrow \mathtt{y.2}\ ]\rangle \\
&\qquad\qquad \mid \mathtt{y} \leftarrow \mathtt{nestjoin}\{\lambda \mathtt{y}\ \mathtt{z}.p_2\ \mathtt{y.1}\ \mathtt{y.2}\ \mathtt{z}\}\ \mathtt{x.2}\ \mathtt{zs}\ ]\rangle \\
&\quad \mid \mathtt{x} \leftarrow \mathtt{nestjoin}\{p_1\}\ \mathtt{xs}\ \mathtt{ys}\ ]
\end{aligned}
$$

The nestjoin{} combinator provides bindings for the outer variable x to the inner comprehension and enables us to introduce a further nestjoin{} combinator. However, similar to Query Q13, the table reference zs can not be lifted to the top-level. The structure of the $\mathcal{SL}$ plan obtained through shredding is mostly equivalent to Figure 69.

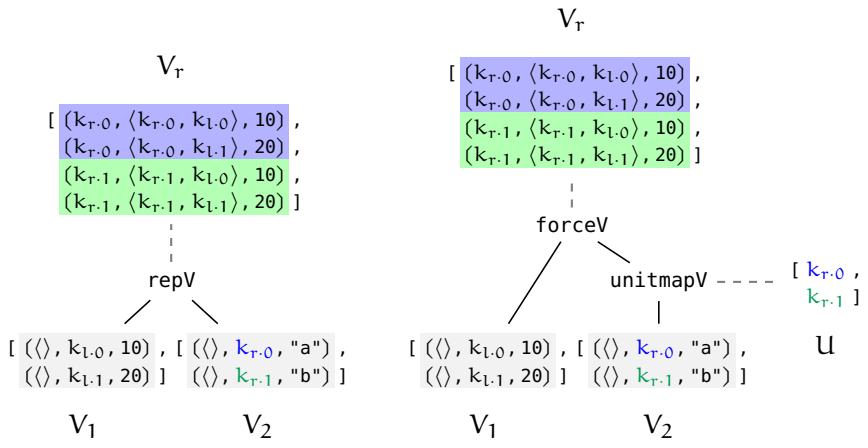
In both examples, variable dependencies between nested comprehensions prevent us from lifting table references out of iteration. Nevertheless, replication is not strictly required to implement the query logic. In this chapter, we sketch a solution to this problem based on a scheme described by Lippmeier *et al.* [Lip+12]. As in earlier chapters, we aim for a solution that does not rely on optimization after the fact, but avoids the problem by construction.

7.1 DELAYING REPLICATION

The $\mathcal{SL}$ operator repV replicates it's left operand as shown in Figure 70a. For each element of the right operand V_2, a copy of the single segment in V_1 is created. The resulting vector has one segment for each element of V_2 and inner index values of V_2 are used to identify those segments. Each of the two segments $k_{r\cdot 0}$ and $k_{r\cdot 1}$ of the result vector V_r maps to the $\langle\rangle$ segment of V_1.

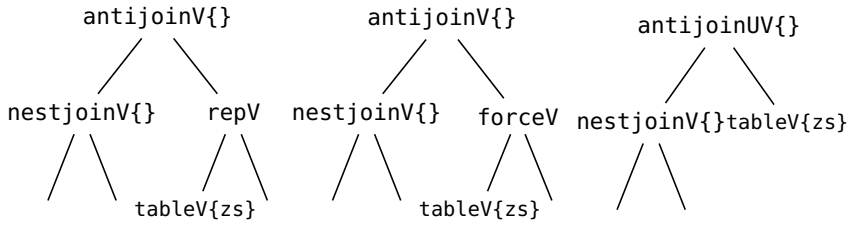
In Figure 70a, this mapping between segments has been *materialized* in the vector V_r. However, V_r can also be accurately described by recording the mapping of segments ($k_{r\cdot 0} \to \langle\rangle$ and $k_{r\cdot 1} \to \langle\rangle$) and the original vector V_1. In this case, we furthermore know that V_1 has only one segment identified by $\langle\rangle$. We record the list of segment identifiers $[k_{r\cdot 0}, k_{r\cdot 1}]$ that map to the $\langle\rangle$ segment and call this list a *unit segment map*. The replication that results in V_r can be described by the segment map combined with V_1. We call a pair (U, V) of a unit segment map U and a segment vector V a *delayed vector*: we only record the information that is necessary to perform the actual replication at a later point.

Consider Figure 70b. The $\mathcal{SL}$ operator unitmapV creates the unit segment map U. To obtain the actual result vector, operator forceV combines the segment map U and the segment vector V_1 and *materializes* the delayed vector. Note that forceV combines the new outer index and the inner index of V_1 to derive unique inner indexes for V_r that match those in Figure 70a.



(a) Eager replication (b) Delayed Replication

Figure 70: Delaying and materializing segment replication.



(a) Eager Replication (b) Delayed Replication (c) Shortcut Replication

Figure 71: Exploiting delayed replication with join operators.

7.1.1 *Exploiting Delayed Replication*

Delaying replication does not improve the situation immediately. The work performed by `forceV` is equivalent to the work performed by `repV`. If each delayed vector has to be materialized right away, we gain nothing. However, we can exploit delaying replication into the pair of segment map and segment vector when performing operations on segment vectors:

- $\mathcal{SL}$ operators that consider individual segments (*e.g.* `distinctV`) or individual rows (*e.g.* `projectV{}`) of a single vector can be applied on the original segment. Only the result of that operation is replicated.
- For some $\mathcal{SL}$ operators — in particular join operators — we can discard the segment map completely and only work on the segment vector.

The former approach clearly saves work. This situation, however, is rare. Note that lifting exclusively replicates table references and list constants (Section 4.2) which are both considered constants. Operations on constant lists are invariant to any enclosing iteration. As described in Section 5.2.3, constant expressions as in `[ distinct ys | x ← xs ]` are hoisted out of the iteration. Hoisting has a similar effect to operating on a delayed vector.

The second approach is more profitable. Recall the replication problem related to the lifted `antijoin{}` combinator in Query Q13. Figure 71a shows the relevant excerpt of the vector plan. Replication of `zs` is necessary to create a vector that matches the segment structure of the vector resulting from

`nestjoinV{}`. The segment join `antijoinV{}` combines matching segments from its inputs.

If we delay the replication of `zs` (Figure 71b), we are forced to materialize the vector before applying `antijoinV{}`. Materialization is required to provide vectors with the same segment structure (*i.e.* compatible outer indexes) to `antijoinV{}` — delaying replication in this example is pointless. However, from the delayed vector we can infer that every segment in the right input of `antijoinV{}` maps to a single segment identified by $\langle\rangle$. Matching segments in `antijoinV{}` is crucial if those segments are actually different. In this scenario, however, each segment of the left operand is joined effectively with the exact same segment of the right operand.

To exploit this, we replace `antijoinV{}` with the `antijoinUV{}` operator in Figure 71c. The latter operator assumes that its right operand only has the $\langle\rangle$ segment and joins each segment from the left operand with that $\langle\rangle$ segment. This allows us to shortcut the replication of `zs`: instead of materializing the vector, we discard the segment map and use the original table reference directly.

7.2 SHREDDING WITH DELAYED VECTORS

In this section, we sketch shredding with delayed replication. During shredding, we distinguish materialized from delayed vectors. New shredding rules for list combinators take this information into account and target specialized $\mathcal{SL}$ operators like `antijoinUV{}`.

To support delayed vectors, we introduce a small number of new $\mathcal{SL}$ operators. We only define the semantics of those operators but do not describe their lowering to the multiset algebra $\mathcal{MA}$. All new operators are sufficiently simple such that their lowering is obvious.

7.2.1 *Delayed Replication*

Shredding with delayed replication requires only minimal changes to our notation. As before, we let V and I range over data vectors and index transformations, respectively. In addition, we let U range over unit segment maps. During shredding, we track vectors as pairs (X, V) of an indicator X and an actual segment vector V. The indicator $\mathcal{M}$ signals that the vector is materialized. Alternatively, the indicator $\mathcal{D}\,U$ indicates that the vector is delayed with segment map U. Hence, a delayed vector is written as $(\mathcal{D}\,U, V)$ and a materialized vector as $(\mathcal{M}, U)$

Rule SHRED-DIST-DELAY handles the $\mathcal{FL}\ \otimes$ combinator and implements delayed replication as discussed in the previous section. We know that e_1 is a top-level list with a single segment. We employ the `unitmapV` operator to derive a segment map from the materialized right operand. The result's inner vector is delayed and consists of the segment map U and the segment vector V_1 obtained for e_1.

$$\textsc{shred-dist-delay}\quad \frac{\Gamma \vdash e_1 \ \wr\ [\delta]^{(\mathcal{M},V_1)} \qquad \Gamma \vdash e_2 \ \wr\ [\rho]^{(\mathcal{M},V_2)} \qquad \begin{bmatrix} V_o \leftarrow \texttt{projectV\{}\lambda x.\langle\rangle\texttt{\}}\ V_2 \\ U \leftarrow \texttt{unitmapV}\ V_2 \end{bmatrix}}{\Gamma \vdash e_1 \otimes e_2 \ \wr\ [[\delta]^{(\mathcal{D}\,U,V_1)}]^{(\mathcal{M},V_o)}}$$

The previous rule assumes that the inner vector for e_2 is materialized. Rule SHRED-DIST-UPDATE, on the other hand, handles the case in which that

$$\frac{\Gamma \vdash V : \mathsf{D}\,(\alpha,\beta,\gamma)}{\Gamma \vdash \texttt{unitmapV}\ V : \mathsf{U}\,\beta}\ \mathcal{SL}\text{-TY-UNITMAP} \qquad \frac{\Gamma \vdash V : \mathsf{D}\,(\langle\rangle,\beta,\gamma) \qquad \Gamma \vdash \mathsf{U} : \mathsf{U}\,\alpha}{\Gamma \vdash \texttt{forceV}\ V\ \mathsf{U} : \mathsf{D}\,(\alpha,\langle\alpha,\beta\rangle,\gamma)}\ \mathcal{SL}\text{-TY-FORCE}$$

Figure 72: New $\mathcal{SL}$ operators that create and materialize delayed vectors.

vector is delayed itself. In this case, we obtain a unit segment map U_i in two steps: `forceV` forces the vector for e_2 and thereby provides all indexes for which V_1 has to be replicated. Those indexes are the basis for the segment map obtained with `unitmapV`.

$$\frac{\Gamma \vdash e_1 \wr [\delta]^{(\mathcal{M},V_1)} \qquad \Gamma \vdash e_2 \wr [\rho]^{(\mathcal{D}\,\mathsf{U},V_2)} \qquad \begin{bmatrix} V_o \leftarrow \texttt{projectV\{}\lambda x.\langle\rangle\texttt{\}}\ V_2 \\ V_f \leftarrow \texttt{forceV}\ V_2\ \mathsf{U} \\ \mathsf{U}_i \leftarrow \texttt{unitmapV}\ V_f \end{bmatrix}}{\Gamma \vdash e_1 \otimes e_2 \wr [[\delta]^{(\mathcal{D}\,\mathsf{U}_i,V_1)}]^{(\mathcal{M},V_o)}}\ \text{SHRED-DIST-UPDATE}$$

Rule SHRED-DIST-UPDATE handles the crucial case of chained replication seen in the example in Figure 69: the list e_2 itself is a product of replication. To obtain the correct segment map, we are forced to materialize the right input. However, to exploit delayed replication in this scenario, we are actually only interested in the vector U_1 for e_1 which provides the data that is replicated in the end. If we are able to discard the segment map as sketched in Section 7.1.1 for the `antijoinV{}` operator, we can shortcut the complete chain of replication — the `forceV` operator is not evaluated.

Figure 72 lists $\mathcal{SL}$ typing rules for the involved operators. The list interpretations of these operators are as follows:

$$\mathcal{S}[\![\texttt{unitmapV}\ V]\!]\rho = [\,x.k \mid x \leftarrow \rho(V)\,] \qquad (\mathcal{SL}\text{-UNITMAP})$$

$$\mathcal{S}[\![\texttt{forceV}\ V\ \mathsf{U}]\!]\rho = [\,(u, \langle u, x.k\rangle, x.p) \mid u \leftarrow \rho(\mathsf{U}), x \leftarrow \rho(V)\,] \qquad (\mathcal{SL}\text{-FORCE})$$

These definitions are as expected: `unitmapV` is a simple projection and `forceV` is essentially equivalent to `repV`.

7.2.2 *Index Transformations*

In the description of shredding (Section 4.3.3.1) we use functions like $(\!|-|\!)_-$ to propagate the various kinds of index transformations through a package. These functions need to be adapted for shredding with delayed vectors. The following definition of $(\!|-|\!)_-$ propagates a replication index transform and distinguishes materialized and delayed vectors.

$$\begin{aligned}
&(\!|\pi|\!)_I = \pi \\
&(\!|\langle \ell_i : \rho_i \rangle_{i=1}^n|\!)_I = \langle \ell_i : (\!|\rho_i|\!)_I \rangle_{i=1}^n \\
&(\!|[\rho]^{(\mathcal{M},V)}|\!)_I = [\,(\!|\rho|\!)_{I'}\,]^{(\mathcal{M},V')} \\
&\quad \left[(V', I') \leftarrow \texttt{apprepV}\ I\ V\right] \\
&(\!|[\rho]^{(\mathcal{D}\,\mathsf{U},V)}|\!)_I = [\rho]^{(\mathcal{D}\,\mathsf{U}',V)} \\
&\quad \left[\mathsf{U}' \leftarrow \texttt{repunitV}\ I\ \mathsf{U}\right]
\end{aligned}$$

$$\frac{\Gamma \vdash \mathsf{U} : \mathsf{U}\,\alpha_1 \qquad \Gamma \vdash \mathsf{I} : \mathsf{R}\,\alpha_1\,\alpha_2}{\Gamma \vdash \texttt{repunitV}\ \mathsf{I}\ \mathsf{U} : \mathsf{U}\,\alpha_2}\ \mathcal{SL}\text{-TY-REPUNIT}$$

Figure 73: $\mathcal{SL}$ typing rule for `repunitV`.

The replicating index transform I is either applied to the vector V for a materialized vector $(\mathcal{M}, V)$ or to the segment map U for a delayed vector $(\mathcal{D}\,\mathsf{U}, V)$. Note that in the latter case, changes don't need to be propagated to the element package ρ because V itself does not change. The $\mathcal{SL}$ operator `repunitV` is defined in Figure 73 and has the following list interpretation.

$$\mathcal{S}[\![\texttt{repunitV}\ \mathsf{I}\ \mathsf{U}]\!]_\rho = [\,\mathsf{i.t} \mid \mathsf{i} \leftarrow \rho(\mathsf{I}), \mathsf{u} \leftarrow \rho(\mathsf{U}), \mathsf{i.f} = \mathsf{u}\,] \qquad (\mathcal{SL}\text{-REPUNIT})$$

Propagation for the remaining index transforms $\langle\!|\ -\ |\!\rangle\ \_$ (index map), $\langle\ -\ \rangle\ \_$ (sorting) and $\langle\ -\ \rangle\ \_$ (filter) is straightforward to adapt to delayed vectors in the same fashion. In all cases, the index transform is applied to the segment map while the corresponding vector remains unchanged — index transforms do not materialize a delayed vector.

7.2.3 *Operations on Delayed Vectors*

Shredding for lifted list combinators (*e.g.* $\texttt{sort}^\uparrow$) and lifted scalar operators (*e.g.* $\_.\ell^\uparrow$) that have *one* operand works uniformly on materialized and delayed vectors. The corresponding $\mathcal{SL}$ operators (*e.g.* `sortV`, `projectV{}`) work on individual segments or segment elements and are oblivious to whether those are replicated. As an example, we show the shredding rule for the $\texttt{distinct}^\uparrow$ combinator adapted to delayed vectors. It maps to the `distinctV` operator that considers each segment of the input vector independently.

$$\frac{\Gamma \vdash e \Rightarrow [[\delta]^{(X_i, V_i)}]^{D_o} \qquad [V \leftarrow \texttt{distinctV}\ V_i]}{\Gamma \vdash \texttt{distinct}^\uparrow\ e \Rightarrow [[\delta]^{(X_i, V)}]^{D_o}}\ \text{SHRED-DISTINCT-LIFT}$$

This rule does not inspect the indicator X_i. The $\mathcal{SL}$ operator `distinctV` is applied to the underlying vector V_i regardless of whether it is materialized or delayed. All shredding rules for lifted combinators with only one operand (*e.g.* SHRED-SORT-LIFT, SHRED-RECORD) are adapted in the same fashion to delayed vectors and do not require materialization of the input.

7.2.4 *Joining Delayed Vectors*

For lifted join combinators, we shortcut replication using delayed vectors as described in Section 7.1.1. We discuss the shredding of $\texttt{antijoinV\{\}}^\uparrow$ only, but shredding for the other join combinators is adapted in the same way. Based on whether the left and right operands are materialized or delayed, we have to consider four cases.

- Both operands are materialized. In that case, we employ the regular `antijoinV{}` operator. The shredding rule SHRED-ANTIJOIN-MAT is a straightforward adaptation of SHRED-ANTIJOIN-LIFT.

SHRED-ANTIJOIN-MAT

$$\frac{\Gamma \vdash e_1 \wr [[\rho_1]^{(\mathcal{M},V_{i\cdot1})}]^{D_{o\cdot1}} \quad \Gamma \vdash e_2 \wr [[\rho_2]^{(\mathcal{M},V_{i\cdot2})}]^{D_{o\cdot2}} \quad \left[(V_j, I) \leftarrow \texttt{antijoinV\{p\}}\ V_{i\cdot1}\ V_{i\cdot2}\right]}{\Gamma \vdash \texttt{antijoin\{p\}}^{\uparrow}\ e_1\ e_2 \wr [[(\rho_1)_I]^{(\mathcal{M},V_j)}]^{D_{o\cdot1}}}$$

- The right operand is delayed. In this case, we statically know that we only have to consider one segment identified by $\langle\rangle$ on the right side. We employ the `antijoinUV{}` operator that foregoes the matching of segments. It filters elements from all segments of the left operand based on the single $\langle\rangle$ segment on the right. Note that this rule also covers the case in which *both* operands are delayed.

SHRED-ANTIJOIN-LIFT-UNIT

$$\frac{\Gamma \vdash e_1 \wr [[\rho_1]^{(X_{i\cdot1},V_{i\cdot1})}]^{D_{o\cdot1}} \quad \Gamma \vdash e_2 \wr [[\rho_2]^{(\mathcal{D}\,\mathsf{U},V_{i\cdot2})}]^{D_{o\cdot2}} \quad \left[(V_j, I) \leftarrow \texttt{antijoinUV\{p\}}\ V_{i\cdot1}\ V_{i\cdot2}\right]}{\Gamma \vdash \texttt{antijoin\{p\}}^{\uparrow}\ e_1\ e_2 \wr [[(\rho_1)_I]^{(X_{i\cdot1},V_j)}]^{D_{o\cdot1}}}$$

The `antijoinUV{}` operator (typing rule in Figure 74) is actually a simplification of the regular `antijoinV{}` operator. In its list interpretation, we merely have to eliminate the additional join predicate that matches segments from Equation ($\mathcal{SL}$-ANTIJOIN).

$$\begin{aligned}&\mathcal{S}[\![\texttt{antijoinUV\{p\}}\ V_1\ V_2]\!]_\rho = \\ &\quad \mathsf{let}\ V = [\, x \mid x \leftarrow \rho(V_1), \mathsf{and}\, [\, \neg\ (p\ x.p\ y.p) \mid y \leftarrow \rho(V_2)\,] \\ &\qquad I = [\, x.k \mid x \leftarrow V\,] \\ &\quad \mathsf{in}\ (V, I)\end{aligned}$$

($\mathcal{SL}$-ANTIJOINU)

Based on this definition, the operator maps directly to a $\mathcal{MA}$ antijoin $\triangleright\{\}$.

- *Only* the left operand is delayed. In this case, care is necessary: we must *not* shortcut the replication for the left operand. Consider the following variant of our motivating example Query Q13 that switches the operands of the `antijoin{}` combinator[1]. In each iteration, elements of `zs` are filtered based on another inner list produced by `nestjoin{}`.

```
[ [ z | y ← antijoin{p2} zs xy.2 ]
| xy ← nestjoin{p1} xs ys ]
```

By shredding, we obtain a delayed vector for the left operand and a materialized vector produced by `nestjoinV{}` for the right operand. Employing `antijoinUV{}` here would be wrong: we do not statically know that all segments on the right are identical — in general, they are not. Here, we actually have to provide copies of `zs` that are filtered based on the corresponding segment on the right. Hence, we have to materialize the left operand and employ the regular `antijoinV{}` combinator.

1 This example reflects the structure of *Example Query 6* given by Steenhagen *et al.* [Ste+94]

$$\textsc{SL-ty-antijoin-unit}\quad \frac{\Gamma \vdash V_1 : \mathsf{D}\,(\alpha, \beta_1, \gamma_1) \qquad \Gamma \vdash V_2 : \mathsf{D}\,(\langle\rangle, \beta_2, \gamma_2) \qquad \vdash s : \gamma_1 \to \gamma_2 \to \mathtt{Bool}}{\Gamma \vdash \mathtt{antijoinUV\{s\}}\ V_1\ V_2 : (\mathsf{D}\,(\alpha, \beta_1, \gamma_1), \mathsf{F}\,\beta_1)}$$

Figure 74: Typing rule for `antijoinUV{}`.

$$\textsc{shred-antijoin-lift-force}\quad \frac{\Gamma \vdash e_1 \wr [\![\rho_1]\!]^{(\mathcal{D}\,U, V_{i\cdot 1})}{}^{\mathsf{D}_{o\cdot 1}} \qquad \Gamma \vdash e_2 \wr [\![\rho_2]\!]^{(\mathcal{M}, V_{i\cdot 2})}{}^{\mathsf{D}_{o\cdot 2}} \qquad \begin{bmatrix} V_f & \leftarrow \mathtt{forceV}\ V_{i\cdot 1}\ U \\ (V_j, I) & \leftarrow \mathtt{antijoinV\{p\}}\ V_f\ V_{i\cdot 2} \end{bmatrix}}{\Gamma \vdash \mathtt{antijoin\{p\}}^{\uparrow}\ e_1\ e_2 \wr [\![(\rho_1)_I]\!]^{(X_{i\cdot 1}, V_j)}{}^{\mathsf{D}_{o\cdot 1}}}$$

This case distinction extends to the other join combinators. Rules for combinators `semijoin{}`$^{\uparrow}$, `groupjoin{}`$^{\uparrow}$ and `nestjoin{}`$^{\uparrow}$ are symmetrical to the ones discussed above. For all of them we materialize the left operand if it is delayed. Note, however, that `thetajoin{}`$^{\uparrow}$ does not have the restriction discussed in the last case: here, either of the operands or both may be delayed.

7.2.5 *Forcing Vectors*

For some combinators, we have no choice but to materialize delayed input vectors. In particular, this is necessary for the following categories:

- Combinators whose $\mathcal{SL}$ implementation requires to relate outer and inner vectors based on their indexes (*e.g.* `concat`$^{\uparrow}$, `reduce{_}`$^{\uparrow}$).
- Combinators whose $\mathcal{SL}$ implementation aligns vectors based on their inner indexes. This category includes lifted scalar operations ($c(\dots)^{\uparrow}$) and record construction.

For these combinators, we adapt shredding rules to materialize delayed inputs with `forceV`. As an example, consider the shredding rule for `reduce{_}`$^{\uparrow}$ if the inner vector is delayed.

$$\textsc{shred-agg-lift-delayed}\quad \frac{\Gamma \vdash e \wr [\![\rho]\!]^{(\mathcal{D}\,U, V_i)}{}^{(X_o, V_o)} \qquad \vdash sz : \delta_a \qquad \begin{bmatrix} V_f & \leftarrow \mathtt{foldV\{}s_z, s_f\mathtt{\}}\ V_i \\ V_m & \leftarrow \mathtt{forceV}\ U\ V_f \\ V_u & \leftarrow \mathtt{unboxV\{}s_z\mathtt{\}}\ V_o\ V_f \\ V & \leftarrow \mathtt{projectV\{\lambda x.x.2\}}\ V_u \end{bmatrix}}{\Gamma \vdash \mathtt{reduce\{}s_z, s_f\mathtt{\}}^{\uparrow}\ e \wr [\delta]^{V}}$$

7.2.6 *Example*

Figure 75 shows the $\mathcal{SL}$ plan obtained by shredding our motivating example Query Q13 with delayed replication. Originally, both operands of `nestjoin{}`$^{\uparrow}$ shred to delayed vectors. We force the left input for `xs` but employ `nestjoinUV{}` and shortcut replication for the right input. The result of `nestjoinUV{}` is materialized. Hence, when shredding `antijoin{}`$^{\uparrow}$,

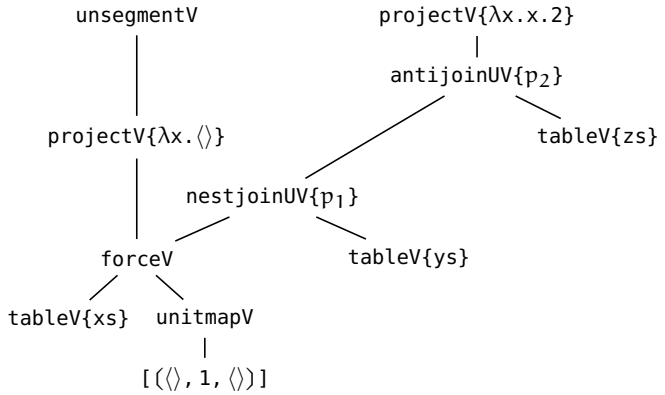


Figure 75: With delayed replication, Query Q13 shreds into this $\mathcal{SL}$ plan.

the left operand is materialized while the right operand is delayed. Again, we employ the `antijoinUV{}` combinator and shortcut the replication of the right operand.

Compared to the original plan in Figure 69, this plan is a substantial improvement. We retain only the trivial replication for `zs` which is easily removed after lowering to a $\mathcal{MA}$ plan. The problematic replication of `zs`, on the other hand, has been eliminated.

7.3 RELATED WORK

The flat representation of nested arrays due to Blelloch and Sabot [BS89] is not capable of sharing segment data. All transformations on the segment structure have to be materialized. Lippmeier *et al.* [Lip+12] introduce a level of indirection for the length-based representation of nested arrays (Section 4.3.1) that allows to share segment data logically. Segment descriptors contain the length of *virtual* segments. Virtual segments map onto *physical* segments that may be shared among multiple virtual segments. All operations on nested arrays are performed on this representation.

We adopt their solution and transfer it to our setting of index-based vectors. Our target data model is more restricted, though. Lippmeier *et al.* allow the physical segments of a vector to be scattered across multiple non-contiguous data blocks. This is not possible in our setting: all segments must map to a single flat multiset or relation in the end. With index-based vectors — in particular *natural* indexes — we are also obliged to maintain the index relationship across vectors. The type of indexes has to be kept consistent which forces materialization of vectors in certain cases.

The approach described by Lippmeier *et al.* is more general than ours. We only consider the special case of replicating top-level lists or base tables: *unit* segment maps describe a mapping to a single segment. Shredding can be further extended to delay not only replication of base tables but index transforms of the segment structure in general. Then, all index transforms (*e.g.* replication, sorting or filtering of segments in inner vectors) are not applied eagerly. Instead, subsequent index transforms can be *combined* into generalized segment maps and the original inner vector preserved. We do not follow this path in this thesis but leave it for future work. The optimizations described in Chapter 5 already eliminate most occurrences of replication in typical queries. Delayed replication in addition provides the freedom

to employ join combinators in comprehensions that are applied iteratively without suffering from replication.

8 EXPERIMENTAL EVALUATION

In preceding chapters we have argued that *Query Flattening* derives efficient relational queries. To support this claim, we investigate the quality of SQL queries generated by an implementation of *Query Flattening*. We take an exemplary qualitative look at relational plans produced for two example queries that make use of different DSH features (Section 8.2). In a subsequent quantitative assessment we investigate the actual run time of generated relational queries. First, we compare the performance of flat-to-flat queries that have nested intermediate results with hand-written SQL code (Section 8.4). This experiment is based on the standard TPC-H benchmark queries. Second, we compare the performance of flat-to-nested queries translated by *Query Flattening* and *Query Shredding* (Section 8.5).

8.1 IMPLEMENTATION OF QUERY FLATTENING

The experimental evaluation is based on an implementation of *Query Flattening* in DSH (Section 1.3). The new backend replaces the *Loop-Lifting* backend of DSH [Gio+11a]. We implemented the backend as a direct — almost literal — implementation of the transformations described in preceding chapters. DSH queries are first translated into $\mathcal{CL}$ and optimized as described in Chapter 5. We implemented $\mathcal{CL}$ optimizations as a system of rewrite rules using the KURE library [SFG14] for strategic programming in Haskell. Optimized $\mathcal{CL}$ queries are subsequently transformed into flat $\mathcal{SL}$ plans exactly as described in Chapter 4. Lowering to $\mathcal{MA}$ plans directly implements the translation rules of Chapter 6. $\mathcal{SL}$ plans are subject to a set of simple optimizations (Section 6.4). Our implementation performs only basic rewrites on $\mathcal{MA}$ plans, mostly related to inlining and merging of projection operators. In $\mathcal{SL}$ and $\mathcal{MA}$ plans we capture sharing of sub-plans and generate directed acyclic graphs of operators. Finally, we flatten records in $\mathcal{MA}$ plans and a code generator similar to the one described by Mayr [May13] translates to SQL:2003 queries. The code generator exploits the DAG structure of plans and binds shared intermediate results as *Common Table Expressions* (CTE).

Each generated SQL:2003 query includes a `ORDER BY` clause. As described in Chapter 6, we encode list element order in relational queries using order labels. The SQL queries generated from $\mathcal{MA}$ plans sort by the combination of outer index and order label. As a consequence, query results arrange segment elements consecutively in the correct list order. This enables the DSH runtime to assemble nested lists from flat relational results by scanning the flat query result only once. The effort of turning logical order labels into a physical order is performed by the database system as part of query execution.

8.2 QUALITY OF RELATIONAL PLANS

With *Query Flattening*, our goal is to translate high-level list-based queries into *idiomatic* relational queries. In particular, the use of abstractions and the complex data model should not negatively impact query runtime. At the same time, the lowering to relational queries should be *comprehensible*.

In preceding chapters we traced the translation of our running example through *Query Flattening* and inspected the resulting $\mathcal{MA}$ plan. Before we go into the experimental evaluation, we exemplarily look at the quality of two more query plans produced by *Query Flattening*. We reuse a scenario used by Rittinger [Rit11] to evaluate the quality of relational plans produced by *Loop-Lifting*. Two queries answer the same information need based on the TPC-H schema: *list the order items of the order with the most parts*. Rittinger describes two approaches to answer this request, both of which we transcribe to DSH queries.

The first variant `db` is written in the style of a relational query and uses two aggregates and a self join to identify the lineitems of those orders that have the maximum number of parts.

```
quantPerOrder :: Q [(Integer, Decimal)]
quantPerOrder = [ (fst g, sum $ map l_quantityQ $ snd g)
                | g <- groupWithKey l_orderkeyQ lineitems ]

db :: Q [LineItem]
db = [ li | li <- lineitems,
            g <- quantPerOrder,
            l_orderkeyQ li == fst g,
            snd g == maximum $ map snd quantPerOrder ]
```

The second variant `prog` is written in the style of a typical Haskell list program and exploits the ordered nature of DSH lists as well as the nested data model[1].

```
prog :: Q [LineItem]
prog = concat $ take 1
              $ sortWith (\g -> -1 * (sum $ map l_quantityQ g))
              $ groupWith l_orderkeyQ lineitems
```

It constructs the nested list of lineitems per order by grouping, sorts the groups in descending order of the number of parts and returns the top group. The `take` combinator takes a 1-element sublist from the beginning of the list of groups which in this case is the group with the highest quantity. Arguably, both queries `prog` and `db` encode reasonable strategies to answer the question.

After translation to $\mathcal{CL}$, both queries are subject to $\mathcal{CL}$ optimizations that result in the queries of Figure 76. In variant `db`, the `maximum` aggregate is hoisted to the top-level and a join operator encodes the correlated iteration over lineitems and and the per-order quantities produced by the `sum` aggregate. In both queries, fold-group fusion combines grouping and aggregation into one combinator. Note that in Figure 76a, the `groupagg{}` combinator produces both the groups explicitly (`g.2`) as well as the sum aggregate of each group (`g.3`).

The optimized $\mathcal{CL}$ expressions describe the structure that *Query Flattening* translates to relational plans. *Lifting* and *shredding* result in the optimized $\mathcal{SL}$ vector plans of Figures 77 and 78 which map the query structure to flat vectors. For query `db`, the $\mathcal{SL}$ plan mirrors the macro structure of the $\mathcal{CL}$ query: the segment join operator `thetajoinV{}` links two vectors obtained from the lineitems table. The original query makes no use of nested lists that would mandate a representation with outer and inner vectors during shred-

1 Note that the two queries are not strictly equivalent: the first query actually computes the items of *all* orders that have the maximum number of parts.

```
let m = maximum [ g.3 | g ← groupagg{0, λs x.s + x.quant} [ ⟨l, l.ok⟩
                                                          | l ← ls ]]
in [ x.1
   | x ← thetajoin{λx y.x.ok = y.1}
           ls
           [ g | g ← groupagg{0, λs x.s + x} [ ⟨l, l.ok⟩ | l ← ls ]
                 g.3 = m ]]
```

(a) $\mathcal{CL}$ query for `db`.

```
concat [ n.1
       | n ← number (sort [ ⟨g.2, -1 * g.3⟩
                          | g ← groupagg{0, λs x.s + x} [ ⟨l, l.ok⟩
                                                        | l ← ls ]]
         n.2 = 1 ]
```

(b) $\mathcal{CL}$ query for `prog`.

Figure 76: $\mathcal{CL}$ representation of DSH queries `db` and `prog` after $\mathcal{CL}$ optimizations (Chapter 5) have been applied.

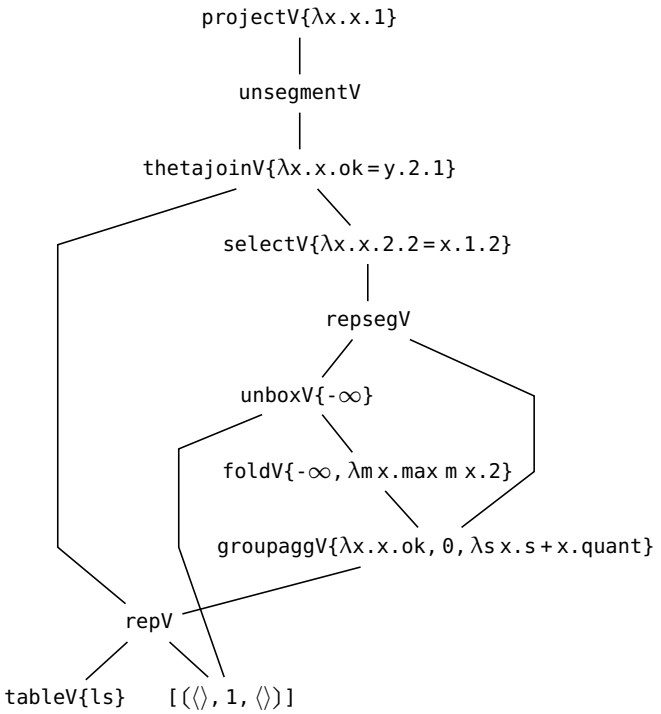


Figure 77: Optimized $\mathcal{SL}$ plan for DSH query `db`.

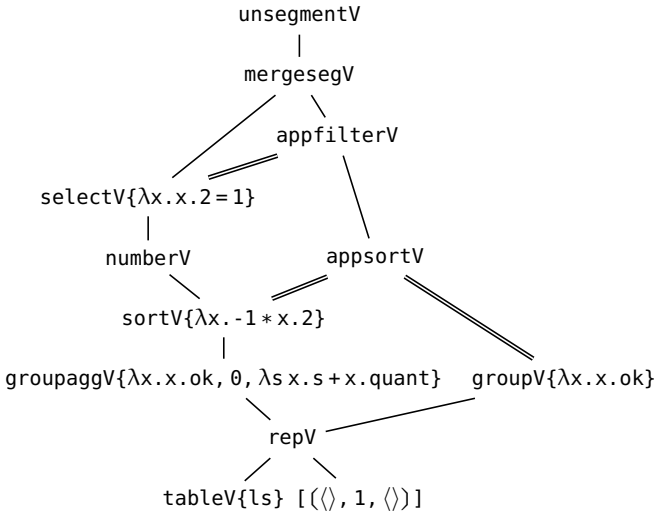
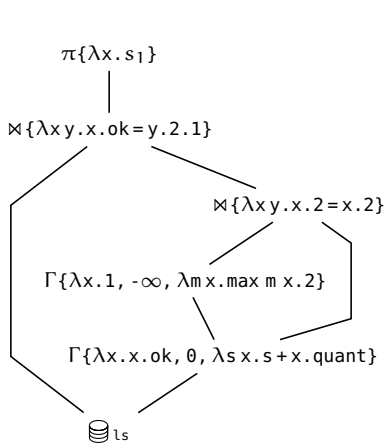


Figure 78: Optimized $\mathcal{SL}$ plan for DSH query `prog`.

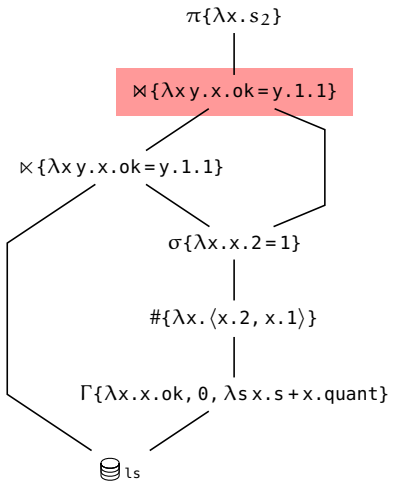
ding. The only nesting introduced is caused by the replication of lineitems over the dummy list `[⟨⟩]` due to desugaring.

Similarly, the plan in Figure 78 directly reflects the structure of the $\mathcal{CL}$ query in Figure 76b. Here, however, the original query makes use of a nested intermediate list and the $\mathcal{SL}$ plan represents it with a pair of vectors produced by `groupaggV{}` and `groupV{}`. This plan exemplarily demonstrates how $\mathcal{SL}$ plans maintain the consistency of vectors that represent nested lists. In the original query, a nested list of groups is transformed by sorting, enumerating and filtering the elements. This is clearly visible in the $\mathcal{SL}$ plan. The left side of the plan reflects the modifications to the outer list. The corresponding vector is sorted (`sortV{}`), enumerated (`numberV`) and filtered (`selectV{}`). In the vector model, sorting and filtering affects segments of the inner vector that contains the elements of each group. These changes are propagated to the inner vector after each step (`appsortV`, `appfilterV`).

We lower the $\mathcal{SL}$ plans of Figures 77 and 78 to the $\mathcal{MA}$ plans shown in Figure 79. Both $\mathcal{MA}$ plans directly express the intent of the original DSH queries and encode almost no overhead for the maintenance of order or nesting. Only the join operator marked red in Figure 78 is redundant and could be removed by further relational rewrites based on functional dependencies. As for our running example in Figure 68, order labels and indexes that establish the complex data model are restricted to a projection at the top of the plan. Simple relational $\mathcal{MA}$ rewrites eliminate literal single-element multisets as well as joins with constant predicates and merge projections. Due to our encoding of indexes and order labels, the $\mathcal{MA}$ plans actually feature less operators than the original $\mathcal{SL}$ plans. Observe that the $\ltimes${} operator in Figure 79b implements the `appfilterV` $\mathcal{SL}$ operator. The operators `groupV{}` and `appsortV`, however, do not reflect in the plan at all. This is not due to a complex optimization scheme but can be directly inferred from the corresponding $\mathcal{MA}$ translation rules (Section 6.3). For `appsortV`, no $\mathcal{MA}$ operators are emitted because we do not rely on encoding the order of segments ($\mathcal{MA}$-APPSORT). Producing the inner vector of the `groupV{}` operator emits only projections ($\mathcal{MA}$-GROUP) which are merged with other downstream projections.



(a) $\mathcal{MA}$ plan for `db`.



(b) $\mathcal{MA}$ plan for `prog`.

Figure 79: Optimized $\mathcal{MA}$ plans for `prog` and `db` (top-most projection expressions s_1 and s_2 omitted for space reasons).

Both $\mathcal{MA}$ plans are reasonable *idiomatic* relational plans for the respective queries. For this reason, we omit an explicit experiment on query runtime. Any relational database system can be expected to cope with these queries.

8.3 SETUP FOR EXPERIMENTS

For our experimental evaluation we use a machine with 60GB of main memory and two quad-core Intel X5570 processors. SQL queries were executed on the database system HyPer v0.5-588-g7dc9661 [Neu11]. HyPer by default makes use of all main memory and all CPU cores. All data sets used in experiments fit into main memory.

In all experiments, we are only interested in the quality and hence execution time of generated SQL queries. We do not consider translation time in DSH nor time spent in reconstruction of the list result. We report end-to-end time for query execution in the database system starting with the submission of the SQL query. Time for query translation is arguably relevant for real-world usage of a language-integrated query system. However, the analytical queries we focus on in our analysis are clearly dominated by query execution time.

8.4 COMPLEX FLAT-TO-FLAT QUERIES

DSH can be used to formulate regular flat-to-flat queries like TPC-H *Q22*. In Chapter 1 we advocate composing complex queries from small building blocks. The resulting flat-to-flat DSH queries (*e.g.* Figure 4) extensively feature nested intermediate results. With a first experiment we investigate whether high-level querying in DSH incurs a runtime penalty when backed by *Query Flattening*.

We implemented all 22 queries of the TPC-H benchmark in DSH in the style of Figure 4, making heavy use of abstractions and nested intermediate results. DSH translates each query into a single SQL:2003 query. We compare the runtime of generated queries with the runtime of the corresponding standard TPC-H benchmark SQL query. Queries are executed on

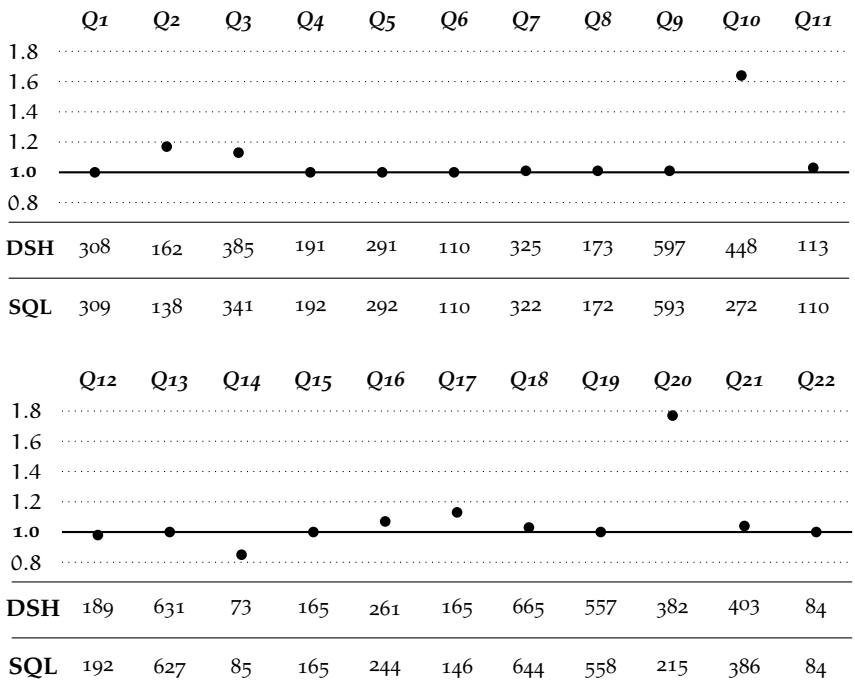


Figure 80: DSH implementation of TPC-H benchmark queries compared with standard SQL benchmark queries. Displayed is the normalized execution time of DSH-generated SQL queries over the execution time of SQL benchmark queries (baseline **1.0**). Also displayed for each query is the absolute execution time (in milliseconds), averaged over 10 runs after one warmup run.

a TPC-H instance with scale factor 10 by the HyPer RDBMS. In addition to the standard indexes on primary- and foreign-key columns, we created indexes on all columns that appear in predicates.

Figure 80 lists the execution time of SQL TPC-H benchmark queries and their DSH equivalent. For each query, we record the average over 10 runs after one discarded warmup run. In addition, Figure 80 shows the normalized execution time of queries translated by DSH, relative to the execution time of the standard SQL queries.

Except for *Q10* and *Q20*, all DSH queries are on par with the standard benchmark SQL queries. This is not surprising: inspection of the SQL:2003 queries generated by DSH shows that they typically match hand-written queries quite closely. As an example, Figure 81 shows the query generated by DSH for TPC-H *Q22*: it has the same structure as the benchmark SQL query and differs only in details. Differences in query execution time seem mostly to be caused by minor variations in the formulation of scalar expressions. The runtime of *Q6*, for example, improved substantially — to the point of matching the runtime of the native SQL query — once we modified the SQL code generator to use the `BETWEEN` expression in a range predicate instead of two independent comparisons.

The runtime of *Q10* and *Q20* in DSH, however, is substantially worse than their SQL counterparts. For *Q10* this is due to differing SQL formulations of the queries' *top-k* part. The query logic of *Q10* limits the result to the first 20 rows in the order of the `ORDER BY` clause. In the standard SQL query, *top-k* is implemented using the `LIMIT` clause. HyPer fuses `LIMIT` and `ORDER BY`: result rows are directly emitted from the sort operator which

```
SELECT (substr(a4.c_phone, 1, 2)) AS c_3_1_x,
       (substr(a4.c_phone, 1, 2)) AS c_4_1_x, COUNT(*) AS c_4_2_x,
       SUM(a4.c_acctbal) AS c_4_3_x
FROM ( VALUES (0)
     ) AS a0(c_2_x),
     ( SELECT 0 AS c_2_1_x, AVG(a1.c_acctbal) AS c_2_4_x
       FROM customer AS a1
       WHERE ((substr(a1.c_phone, 1, 2))
              IN ( VALUES ('13'), ('31'), ('23'),
                          ('29'), ('30'), ('18'), ('17')))
             AND (a1.c_acctbal > CAST(0.0 AS DECIMAL(1,1)))
     ) AS a3(c_2_1_x, c_2_4_x),
     customer AS a4
WHERE ((substr(a4.c_phone, 1, 2))
       IN ( VALUES ('13'), ('31'), ('23'), ('29'),
                   ('30'), ('18'), ('17')))
      AND ((NOT (a4.c_custkey IN ( SELECT a5.o_custkey
                                   FROM orders AS a5))))
      AND (a4.c_acctbal > a3.c_2_4_x)
      AND (a0.c_2_x = a3.c_2_1_x)
GROUP BY (substr(a4.c_phone, 1, 2))
ORDER BY c_3_1_x ASC;
```

Figure 81: SQL query generated for the DSH formulation of *Q22*.

stops once it has produced 20 rows. In contrast, DSH emits a combination of the `row_number()` window function and a selection. For the DSH query, HyPer sorts the large intermediate result to implement the window function, selects the first 20 rows explicitly and subsequently sorts the final result after selection.

For *Q20*, the DSH query is noticeably slower as well. Upon closer inspection, this slowdown is caused by a semantic difference in the treatment of aggregates on empty inputs. *Q20* contains the following predicate with a correlated subquery:

```
... (SELECT ps_suppkey
     FROM partsupp
     WHERE ...
           AND ps_availqty > (
             SELECT 0.5 * SUM(l_quantity)
             FROM lineitem
             WHERE l_partkey = ps_partkey AND ...))
...
```

Assume that for a given row from `partsupp`, the correlated subquery returns an empty result. In line with the SQL semantics, the `SUM` aggregate function and hence the subquery itself returns `NULL` and the row is skipped. Effectively, this predicate has the secondary effect of skipping all suppliers that do not supply any lineitems in the selected time interval.

In the DSH formulation of *Q20*, combinator `sum` is used to implement the predicate. However, in contrast to the SQL aggregate, `sum` on an empty list is not *unknown* (`NULL`) but evaluates to zero. This behaviour not only reflects the regular Haskell `sum` combinator but is arguably more reasonable and less surprising. For *Q20*, however we have to be careful not to include suppliers in the result that supply no lineitems — for those, the predicate

on ps_availqty would be fulfilled since sum evaluates to zero. In the DSH implementation of *Q20* we explicitly exclude such suppliers with an additional predicate on partsupp. In the following excerpt, we explicitly check whether the current partsupp row is referenced by lineitems in the relevant interval (Line 10):

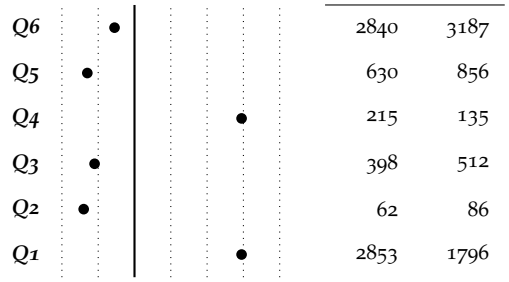
```
excessBoundary :: Interval -> Q PartSupp -> Q Decimal
excessBoundary interval ps =
    0.5 * sum (stockQuantities interval ps)

excessSuppliers :: Text -> Interval -> Q [Integer]
excessSuppliers color interval =
    [ ps_suppkeyQ ps
    | ps <- partsupps,
      ...,
      not (null (stockQuantities interval ps)),
      ps_availqtyQ ps > excessBoundary interval ps ]
```

We have argued in Chapter 1 that preserving the semantics of the host language precisely is essential for language-integrated querying. Preserving the semantics of the Haskell sum combinator does not come for free in this example, though. The query generated by DSH has to identify empty inputs to sum and add the value zero for those. This work is performed by the groupjoin{} combinator used to unnest the correlated subquery (Section 5.2). The relational implementation of the groupjoin{} combinator relies on a left outerjoin to identify empty groups. The resulting relational execution plan for HyPer is more expensive than the plan for the standard SQL form of *Q20* which does not require an outerjoin. Furthermore, the additional check in Line 10 maps to a semijoin and has a runtime cost. We verified that these differences are the sole cause of the slowdown for *Q20* by manually modifying the generated query.

8.5 NESTED QUERIES

In a second experiment, we investigate the performance of flat-to-nested queries translated by DSH. In the experimental evaluation of *Query Shredding*, Cheney *et al.* use six flat-to-nested benchmark queries *Q1* to *Q6* [CLW14a]. These queries return nested results with a nesting depth of one to four, using the query constructs supported by *Query Shredding*: nested comprehensions, bag union and quantifiers expressed with a bag emptiness test. We refer to Cheney *et al.* for a detailed description of the benchmark queries. Their evaluation compares the end-to-end execution time of those queries in Links with *Query Shredding* and *Loop-Lifting* (using the Pathfinder compiler [Rit11]) backends. Results by Cheney *et al.* demonstrate that *Query Shredding* generates reasonable SQL queries that compare favorably to those generated by *Loop-Lifting*. *Loop-Lifting* queries do not finish in reasonable time for queries *Q1* and *Q6* and perform substantially worse for *Q2*.

We use benchmark queries *Q1* to *Q6* to assess the SQL code generated for flat-to-nested queries by *Query Flattening* and *Query Shredding*. In contrast to the evaluation of Cheney *et al.*, we exclusively consider the execution time of the generated SQL queries and do not measure the time spent for query translation or fetching and reconstruction of the host language result values. Implementations of *Query Shredding* in Links and *Query Flattening* in DSH are backed by different database drivers as well as different runtimes imple-



	DSH	Links
Q6	2840	3187
Q5	630	856
Q4	215	135
Q3	398	512
Q2	62	86
Q1	2853	1796

Figure 82: Relative execution times of shredding benchmark queries translated by the flattening compiler on HyPer DBMS (Sum of individual flat queries, average over 10 runs each).

mented in OCaml and Haskell, respectively. Comparing those is unlikely to yield meaningful results. In addition, we can expect that for compute-heavy analytical queries on large datasets, SQL execution time will dominate.

For each of the six nested benchmark query, we obtained the bundle of SQL queries generated by Links with *Query Shredding*. We transcribed the Links formulation of those queries to Haskell using DSH and obtained the bundle of generated SQL queries. We used the data generator by Cheney *et al.* and generated a data set with 4096 departments — the largest data set used in their evaluation. For each nested benchmark query, we execute the bundle of SQL queries 10 times. We report *cumulative* execution times for all queries in a bundle, averaged over ten runs after one discarded warmup run. SQL queries are executed on the HyPer RDBMS.

Figure 82 reports normalized as well as absolute cumulative execution times for *Q1* to *Q6*. Except for queries *Q1* and *Q4*, *Query Flattening* performs slightly better than *Query Shredding*. More importantly, for *Q1* and *Q4* the execution time for *Query Flattening* is higher but still within a reasonable range. In particular, *Query Flattening* does not exhibit the performance problems of *Loop-Lifting* queries reported by Cheney *et al.* [CLW14a].

This is not surprising: the problems with *Loop-Lifting* queries are caused by window functions on top of cartesian products that Pathfinder is not able to remove. As established in Chapter 6, *Query Flattening* does not use window functions for indexes or order labels. Window functions are only used for a limited set of combinators (`append`, `distinct` and `number`). Indeed, among the six benchmark queries, only *Q6* features a `row_number()` window function due to `append` being used.

Recall that *Query Flattening* observes list semantics while *Query Shredding* works with multiset semantics. The sorting effort in SQL queries generated by *Query Shredding*, however is typically at least as high as the sorting effort in the case of *Query Flattening*. *Query Shredding* uses window functions to generate synthetic indexes. In contrast, *Query Flattening* uses natural indexes and order labels that do not rely on window functions. In many queries (*e.g.* benchmark queries *Q1* to *Q5*) the only sorting construct in SQL code generated by *Query Flattening* is the top-level **`ORDER BY`** clause that arranges elements into the correct list and segment order to ease construction of nested list results (Section 8.1).

9 SUMMARY AND OUTLOOK

This thesis takes on the challenge of query flattening: supporting nested query languages with a complex nested and ordered data model on off-the-shelf relational query engines. We have found prior work on this problem to not be fully convincing (Chapter 2). Taken from a seemingly distant domain — implementation of nested data parallelism — the flattening transformation provides the basis to improve on prior approaches. With *Query Flattening* we have revisited all aspects of query flattening: from the translation to flat queries over query optimization to the generation of relational queries. We believe that the work presented in this thesis constitutes a clear step forward on all of these aspects. We have been able to draw on previous work both from the domain of database query languages and query optimization as well as the implementation of data-parallel languages — a clear indicator that these domains are not so distant after all.

9.1 CONTRIBUTIONS

This thesis is not the first attempt to leverage the flattening transformation for query flattening. In unpublished work, Weijers [Wei17] adapts Leshchinskiy's higher-order flattening [Les05] to generate the relational algebra dialect of Pathfinder. The scheme for indexes and order labels described by Weijers is mostly equivalent to that of *Loop-Lifting*. Weijers relies on Pathfinder for query optimization. The work presented in this thesis started by adding optimization to Weijers' translation but evolved over time into a completely separate effort. In contrast to Weijers' work, we argue that higher-order flattening is not necessary since the target language (relational algebra) is first-order (see also Section 1.4.2). Then, considering first-class functions complicates the translation considerably and does not provide additional insights.

In the following, we summarize our contributions.

9.1.1 *Comprehensible Query Flattening*

Arguably, prior descriptions of query flattening suffer either from a lack of expressiveness or from a complex translation (Chapter 2). *Query Flattening* (Chapter 4) combines the support for an expressive query language with a comprehensible, well-structured translation. The lowering to relational queries is structured into a pipeline of simple, clearly confined steps. Each lowering step implements one central aspect of query flattening. *Lifting* provides a description of a calculus query in terms of algebraic bulk operators. *Shredding* implements those operators on flat collections. Translation to $\mathcal{MA}$ maps ordered collections to the unordered relational world. Individual translation steps target a number of intermediate languages that each remove one layer of abstraction. We have defined the static and dynamic semantics of these intermediate languages precisely as a basis for reasoning about the type-safety and correctness of *Query Flattening*.

The fine-grained lowering pipeline is the basis for all other improvements discussed in this thesis. It has enabled us to apply query optimization just

at the right levels of abstraction and devise improved encodings for indexes and element order without changing the whole pipeline.

9.1.2 *Query Optimization*

It is well-established that compositional query flattening leads to inefficient queries that relational engines can not cope with (Section 5.1). Our main result on query optimization in the context of query flattening (Chapter 5) is to show that major problems are not specific to query flattening but well-known from the implementation of orthogonal query calculi. We face optimization challenges that have been extensively researched for complex-object query languages.

With this perspective, we conclude that optimization should be performed before flattening, not after. The structure of queries is best understood in a high-level query language such as $\mathcal{CL}$, not after translation to a low-level language such as relational algebra. Optimizing early allows to employ established optimization techniques (Section 5.2) that integrate well with *Query Flattening* (Sections 5.3 and 5.4).

High-level $\mathcal{CL}$ optimizations fix the macro query structure prior to flattening. Little optimization effort is required after flattening. Here, we profit from the fine-grained lowering pipeline of *Query Flattening*: much of the remaining work (*e.g.* merging of scalar expressions) can be performed at the intermediate level of vector plans where rewrites are easier to express than in relational algebra (Section 4.3.4).

Not all problems however, can be addressed on the level of calculus or algebra plan rewriting. For those remaining issues we have been able to draw on techniques originally developed for the flattening transformation (Chapter 7).

9.1.3 *Relational Code Generator*

Even though high-level optimizations on $\mathcal{CL}$ eliminate the major problematic aspects of flattened queries, the quality of actual backend queries still hinges on an efficient relational encoding of indexes and order labels. In this work, we devised efficient encodings that describe indexes and order in terms of base columns and do not incur the runtime overhead of window functions. In contrast to prior work, we do not rely on post-mortem optimizations here. Our translation to relational queries makes use of the efficient encoding from the get-go without impacting preceding lowering steps.

9.1.4 *Database-Supported Haskell*

With it's expressive library of list combinators and its native support for comprehension syntax, Haskell provides an ideal environment for language-integrated querying. With DSH, Giorgidze *et al.* [Gio+11a; Gio+11b] present a seamless embedding of queries into Haskell. During the work on this thesis we built on their foundation and used DSH as the basis for the research on query flattening. This culminated in a new implementation of DSH that only keeps the language-embedding of DSH and adds a complete new backend based on *Query Flattening*. Our implementation provides all components discussed in this thesis (optimization, SQL code generation *etc.*) and does not rely anymore on the Pathfinder compiler.

The nested data model of DSH enables a high-level style of querying with reusable abstractions. Faithful preservation of list semantics allows to formulate order-aware queries with ease. Our experiments (Chapter 8) indicate that this does not come at the expense of query runtime. DSH reliably generates SQL queries of reasonable quality that typically perform as well as hand-written queries. Hence, for language-integrated queries, DSH provides "abstraction without regret" [Rom12]. The usefulness of DSH is further underlined by its use in research: Stolarek and Cheney [SC18] implemented *language-integrated provenance* based on DSH.

9.2 FUTURE WORK

We believe that the work presented here can and should be extended in multiple directions. Specifically, we see the following opportunities for follow-up research.

- It remains to be shown formally that *Query Flattening* preserves the semantics of the original $\mathcal{CL}$ query. Compared to *Loop-Lifting*, we believe that the starting position is much improved. The fine-grained lowering pipeline allows to check the correctness of a number of small, well-defined steps that focus on specific aspects of query flattening. We have defined the semantics of intermediate languages based on indexed lists. Indexes relate all intermediate forms down to $\mathcal{SL}$ vectors. We conjecture that a correctness proof could be based on the proof given by Cheney *et al.* [CLW14b] for *Query Shredding* that relies on indexes.

  Query Flattening guarantees that a flat-to-flat query in $\mathcal{CL}$ lowers to a single flat relational query. Hence, as a side-effect, a rigorous proof of correctness for *Query Flattening* would establish that $\mathcal{CL}$ is a *conservative extension* of relational algebra. This would extend the conservative extension property to a more expressive query language than in earlier work by Libkin and Wong [LW97].

- We have described *Query Flattening* based on list semantics. Any bag query can be expressed as a list query as well by ignoring the order of the result. As demonstrated in Chapter 8, the overhead for maintaining list order in generated relational queries is small if no explicit order-aware operations like enumeration are used. Still, tracking order columns requires some work that is redundant for queries that do not rely on list semantics.

  Query Flattening, however, does not rely on list semantics and we expect that it extends without problems to multiset semantics. All intermediate languages and the final flat representation are based on indexes that carry no notion of order. This is different to other implementations of the flattening transformation that rely on list order and positions. In *Query Flattening*, order only becomes relevant in the last lowering to $\mathcal{MA}$. Compared to *Query Shredding* (Chapter 2) which is defined on multisets, *Query Flattening* on multisets would still support a more expressive query language (*e.g.* aggregation, grouping, duplicate elimination). Our optimizations on comprehensions are valid not only for lists but also for multiset queries.

  It should be possible to extend *Query Flattening* to queries that work on multisets and lists at the same time. The monad comprehension

calculus [Bun+95; Gru99] provides a basis for query languages that support multiple collection types.

- Recently, Hoelsch *et al.* [HGS16] as well as Cao and Badia [CB07] have proposed the use of nested relations and nested relational algebra for the optimization of relational queries with nested subqueries. Optimization of nested subqueries profits from the ability to explicitly represent nested intermediate results, in particular *empty* nested intermediate results. *Query Flattening* can help in such approaches. As demonstrated in Chapter 5, we can employ unnesting techniques for nested query languages on $\mathcal{CL}$. Unnesting of correlated subqueries with `nestjoin{}` and `groupjoin{}` operators, for example, is safe against the infamous `COUNT` bug [GW87] and similar problems. These techniques can be used in flat-to-flat queries as well, using nested intermediate results. Subsequently, *Query Flattening* supports these unnesting combinators and provides a systematic lowering to a flat query that can be executed on relational database systems.

A

INDEXED SEMANTICS OF LIFTED OPERATORS

$$\mathcal{F}[\![x]\!]_\rho = \rho(x) \qquad (\mathcal{FL}\text{-VAR})$$

$$\mathcal{F}[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!]_\rho = \mathcal{F}[\![e_2]\!]_{\rho[x \mapsto \mathcal{F}[\![e_1]\!]_\rho]} \qquad (\mathcal{FL}\text{-LET})$$

$$\mathcal{F}[\![\langle \ell_1 = e_1, \ell_2 = e_2 \rangle^\uparrow]\!] = [\ \langle k = x.k, p = \langle \ell_1 = x.p, \ell_2 = y.p \rangle \rangle \mid x \leftarrow \mathcal{F}[\![e_1]\!]_\rho, y \leftarrow \mathcal{F}[\![e_2]\!]_\rho, x.k = y.k\ ] \qquad (\mathcal{FL}\text{-RECORD-LIFT})$$

$$\mathcal{F}[\![\texttt{concat } e]\!]_\rho = [\ \langle k = \langle xs.k, x.k \rangle, p = x.p \rangle \mid xs \leftarrow \mathcal{F}[\![e]\!]_\rho, x \leftarrow xs.p\ ] \qquad (\mathcal{FL}\text{-CONCAT})$$

$$\mathcal{F}[\![\texttt{restrict } e]\!]_\rho = [\ \langle k = x.k, p = x.p.1 \rangle \mid x \leftarrow \mathcal{F}[\![e]\!]_\rho, x.p.2\ ] \qquad (\mathcal{FL}\text{-RESTRICT})$$

$$\mathcal{F}[\![\texttt{combine } e_1\ e_2\ e_3]\!]_\rho = [\ \langle k = x.k, p = y.p \rangle \mid x \leftarrow \mathcal{F}[\![e_1]\!]_\rho, y \leftarrow \mathcal{F}[\![e_2]\!]_\rho \ +\!\!+\ \mathcal{F}[\![e_3]\!]_\rho, x.k = y.k\ ] \qquad (\mathcal{FL}\text{-COMBINE})$$

$$\mathcal{F}[\![\texttt{sng}^\uparrow\ e]\!]_\rho = [\ \langle k = x.k, p = [\langle k = \langle\rangle, p = x.p \rangle] \rangle \mid x \leftarrow \mathcal{F}[\![e]\!]_\rho\ ] \qquad (\mathcal{FL}\text{-SNG-LIFT})$$

$$\mathcal{F}[\![\texttt{reduce\{z, f\}}^\uparrow\ e]\!]_\rho = [\ \langle k = xs.k, p = \text{foldl } (f \oslash \pi_p)\ z\ xs.p \rangle \mid xs \leftarrow \mathcal{F}[\![e]\!]_\rho\ ] \qquad (\mathcal{FL}\text{-AGG-LIFT})$$

$$\mathcal{F}[\![\texttt{distinct}^\uparrow\ e]\!]_\rho = [\ \langle k = xs.k, p = \text{nubWith } \pi_p\ xs.p \rangle \mid xs \leftarrow \mathcal{F}[\![e]\!]_\rho\ ] \qquad (\mathcal{FL}\text{-DISTINCT-LIFT})$$

$$\mathcal{F}[\![\texttt{sort}^\uparrow\ e]\!]_\rho = [\ \langle k = xs.k, p = [\ \langle k = x.k, p = x.p.1 \rangle \mid x \leftarrow \text{sortWith } (\pi_1 \circ \pi_p)\ xs.p\ ] \rangle \mid xs \leftarrow \mathcal{F}[\![e]\!]_\rho\ ] \qquad (\mathcal{FL}\text{-SORT-LIFT})$$

$$\mathcal{F}[\![\texttt{concat}^\uparrow\ e]\!]_\rho = [\ \langle k = xss.k, p = [\ \langle k = \langle xs.k, x.k \rangle, p = x.p \rangle \mid xs \leftarrow xss.p, x \leftarrow xs.p\ ] \rangle \mid xss \leftarrow \mathcal{F}[\![e]\!]_\rho\ ] \qquad (\mathcal{FL}\text{-CONCAT-LIFT})$$

$$\mathcal{F}[\![\texttt{group}^\uparrow\ e]\!]_\rho = [\ \langle k = xs.k, p = [\ \langle k = g.1, p = \langle g.1, [\ \langle k = x.k, p = x.p.1 \rangle \mid x \leftarrow g.2\ ] \rangle \rangle \mid g \leftarrow \text{groupWith } (\pi_1 \circ \pi_p)\ xs.p\ ] \rangle \mid xs \leftarrow \mathcal{F}[\![e]\!]_\rho\ ] \qquad (\mathcal{FL}\text{-GROUP-LIFT})$$

$$
\begin{aligned}
&\mathcal{F}[\![\mathsf{append}^{\uparrow}\ e_1\ e_2]\!]_\rho = \\
&\quad [\ \langle \mathsf{k} = \mathsf{xs.k}, \mathsf{p} = [\ \langle \mathsf{k} = \langle 1, \mathsf{x.2}\rangle, \mathsf{p} = \mathsf{x.1.p}\rangle \mid \mathsf{x} \leftarrow \mathsf{enum\ xs.p}\ ] \\
&\qquad\qquad\qquad +\!\!+ \\
&\qquad\qquad\qquad [\ \langle \mathsf{k} = \langle 2, \mathsf{y.2}\rangle, \mathsf{p} = \mathsf{y.1.p}\rangle \mid \mathsf{y} \leftarrow \mathsf{enum\ ys.p}\ ] \\
&\quad \mid \mathsf{xs} \leftarrow \mathcal{F}[\![e_1]\!]_\rho, \mathsf{ys} \leftarrow \mathcal{F}[\![e_2]\!]_\rho, \mathsf{xs.k} = \mathsf{ys.k}\ ]
\end{aligned}
$$

($\mathcal{FL}$-APPEND-LIFT)

$$
\begin{aligned}
&\mathcal{F}[\![\mathsf{combine}^{\uparrow}\ e_b\ e_t\ e_f]\!]_\rho = \\
&\quad [\ \langle \mathsf{k} = \mathsf{xsb.k}, \mathsf{p} = [\ \langle \mathsf{k} = \mathsf{x.k}, \mathsf{p} = \mathsf{y.p}\rangle \\
&\qquad\qquad\qquad \mid \mathsf{xb} \leftarrow \mathsf{xsb.p}, \mathsf{x} \leftarrow \mathsf{x} \leftarrow \mathsf{xst.p}\ +\!\!+\ \mathsf{xsf.p} \\
&\qquad\qquad\qquad , \mathsf{xb.p} = \mathsf{x.p}\ ]\rangle \\
&\quad \mid \mathsf{xsb} \leftarrow \mathcal{F}[\![e_b]\!]_\rho, \mathsf{xst} \leftarrow \mathcal{F}[\![e_t]\!]_\rho, \mathsf{xsf} \leftarrow \mathcal{F}[\![e_f]\!]_\rho \\
&\quad , \mathsf{xsb.k} = \mathsf{xst.k}, \mathsf{xsb.k} = \mathsf{xsf.k}\ ]
\end{aligned}
$$

($\mathcal{FL}$-COMBINE-LIFT)

B

INTRODUCTION RULES FOR JOIN COMBINATORS

We use the following shortcut to substitute two variables with components of a pair:

$$tup_{x,y,u}(e) = (e[u.1/x])[u.2/y]$$
$$trip_{x,u}(e) = e[\langle u.1, u.2\rangle/x]$$

Introduction rules for join combinators are the following:

$$[\ h \mid qs, x \leftarrow xs, y \leftarrow ys, p\ x\ y, qs'\ ]$$
$$\rightsquigarrow$$
$$[\ tup_{x,y,u}(h) \mid qs, u \leftarrow \texttt{thetajoin}\{\lambda x\, y.p\}\ xs\ ys,\ tup_{x,y,u}(qs')\ ] \qquad \text{(THETAJOIN)}$$

$$[\ h \mid qs, x \leftarrow xs, \texttt{and}\ [\ p_q\ y \mid y \leftarrow ys, p_r\ x\ y\ ], qs'\ ]$$
$$\rightsquigarrow$$
$$[\ h \mid qs, x \leftarrow \texttt{antijoin}\{\lambda x\, y.p_r\}\ xs\ [\ y \mid y \leftarrow ys, \neg\ p_q\ ], qs'\ ] \qquad \text{(ANTIJOIN-15)}$$

$$[\ h \mid qs, x \leftarrow xs, \texttt{and}\ [\ p_q\ x\ y \mid y \leftarrow ys, p_r\ x\ y\ ], qs'\ ]$$
$$\rightsquigarrow$$
$$[\ h \mid qs, x \leftarrow \texttt{antijoin}\{\lambda x\, y.p_r \wedge \neg\ p_q\}\ xs\ ys, qs'\ ] \qquad \text{(ANTIJOIN-16)}$$

$$[\ h \mid qs, x \leftarrow xs, \texttt{or}\ [\ p_q\ x\ y \mid y \leftarrow ys, p_r\ x\ y\ ], qs'\ ]$$
$$\rightsquigarrow$$
$$[\ h \mid qs, x \leftarrow \texttt{semijoin}\{\lambda x\, y.p_q \wedge p_r\}\ xs\ ys, qs'\ ] \qquad \text{(SEMIJOIN)}$$

$$\neg(\texttt{or}\ [\ h \mid qs\ ]) \rightsquigarrow \texttt{and}\ [\ \neg h \mid qs\ ] \qquad \text{(NORM-UNIVERSAL)}$$
$$\neg(\texttt{and}\ [\ h \mid qs\ ]) \rightsquigarrow \texttt{or}\ [\ \neg h \mid qs\ ] \qquad \text{(NORM-EXISTENTIAL)}$$

$$[\ h \setminus [\ e \mid qs_1, y \leftarrow ys, s\ x\ y, qs_2\ ] \mid qs_3, x \leftarrow xs, qs_4\ ]$$
$$\rightsquigarrow$$
$$[\ (h \setminus [\ tup_{x,y,v}(e) \mid qs_1, v \leftarrow u.2,\ tup_{x,y,v}(qs_2)\ ])[u.1/x]$$
$$\mid qs_3, u \leftarrow \texttt{nestjoin}\{\lambda x\, y.s\}\ xs\ ys,\ qs_4[u.1/x]\ ]$$
$$\text{(NESTJOIN-HEAD)}$$

$$
\begin{array}{l}
[\ h \mid qs_1, x \leftarrow xs, qs_2, p \searrow [\ e \mid qs_3, y \leftarrow ys, s\ x\ y, qs_4, \ ], qs_5\ ] \\
\quad \rightsquigarrow \\
[\ h[u.1/x] \\
\mid qs_1 \\
, u \leftarrow \texttt{nestjoin}\{\lambda x\, y.s\}\ xs\ ys \\
, qs_2[u.1/x] \\
, (p \searrow [\ \mathit{tup}_{x,y,v}(e) \mid qs_3, v \leftarrow u.2, \mathit{tup}_{x,y,v}(qs_4)\ ])[u.1/x] \\
, qs_5[u.1/x]\ ]
\end{array}
$$

(NESTJOIN-GUARD)

Both Rules NESTJOIN-HEAD and NESTJOIN-GUARD assume that the inner list ys is closed, *i.e.* $fv(ys) = \emptyset$. In practice, we have found it useful to extend both rules to consider predicates $p\ y$ in the inner comprehension that only depend on the inner generator. Such predicates can be evaluated on the right-hand input of the nestjoin operator as $\texttt{nestjoin}\{\lambda x\, y.s\}\ xs\ [\ y \mid y \leftarrow ys, p\ ]$.

$$
\begin{array}{l}
[\ h \searrow (\texttt{reduce}\{z, f\}\ [\ e\ x \mid x \leftarrow g.2\ ])\ \mid qs, g \leftarrow \texttt{group}\ xs, qs'\ ] \\
\quad \rightsquigarrow \\
[\ \mathit{trip}_{g,v}(h \searrow v.3) \mid qs, v \leftarrow \texttt{groupagg}\{s_z, f \oslash e\}\ xs, \mathit{trip}_{g,v}(qs')\ ]
\end{array}
$$

(GROUPAGG-HEAD)

$$
\begin{array}{l}
[\ h \searrow (\texttt{reduce}\{z', f'\}\ [\ e\ x \mid x \leftarrow g.2\ ])\ \mid qs, g \leftarrow \texttt{groupagg}\{z, f\}\ xs, qs'\ ] \\
\quad \rightsquigarrow \\
[\ (h \searrow v.3.2)[\langle v.1, v.2, v.3.1\rangle/g] \\
\mid qs, v \leftarrow \texttt{groupagg}\{\langle z, z'\rangle, f \parallel (f' \oslash e)\}\ xs\ ] \\
, qs'[\langle v.1, v.2, v.3.1\rangle/g]\ ]
\end{array}
$$

(GROUPAGG-HEAD-EXTEND)

$$
\begin{array}{l}
[\ h \searrow (\texttt{reduce}\{z, f\}\ [\ e\ x \mid x \leftarrow g.2\ ])\ \mid qs, g \leftarrow \texttt{nestjoin}\{p\}\ xs\ ys, qs'\ ] \\
\quad \rightsquigarrow \\
[\ \mathit{trip}_{g,v}(h \searrow v.3) \mid qs, v \leftarrow \texttt{groupjoin}\{p, z, f \oslash e\}\ xs\ ys, \mathit{trip}_{g,v}(qs')\ ]
\end{array}
$$

(GROUPJOIN-HEAD)

$$
\begin{array}{l}
[\ h \mid qs, g \leftarrow \texttt{nestjoin}\{p\}\ xs\ ys, qs' \\
, (\texttt{reduce}\{z, f\}\ [\ e\ x \mid x \leftarrow g.2\ ]), qs''\ ] \\
\quad \rightsquigarrow \\
[\ \mathit{trip}_{g,v}(h) \mid qs, v \leftarrow \texttt{groupjoin}\{p, z, f \oslash e\}\ xs\ ys, \mathit{trip}_{g,v}(qs') \\
, \mathit{trip}_{g,v}(p' \searrow v.3), \mathit{trip}_{g,v}(qs'')\ ]
\end{array}
$$

(GROUPJOIN-GUARD)

Similar to Rule GROUPAGG-HEAD-EXTEND, groupjoin operators can be extended with additional aggregates.

BIBLIOGRAPHY

[AB86] Serge Abiteboul and Nicole Bidoit. Non First Normal Form Relations: An Algebra Allowing Data Restructuring. In *Journal of Computer and System Sciences* 33.3 (1986), pp. 361–393.

[Ale+15] Alexander Alexandrov, Lauritz Thamsen, Andreas Kunft, Odej Kao, Asterios Katsifodimos, Tobias Herb, and Volker Markl. Implicit Parallelism Through Deep Language Embedding. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2015.

[AB87] Malcolm P. Atkinson and Peter Buneman. Types and Persistence in Database Programming Languages. In *ACM Computing Surveys* 19.2 (1987), pp. 105–170.

[BR12] Lars Bergstrom and John Reppy. Nested Data-Parallelism on the GPU. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2012, pp. 247–258.

[Ber+13] Lars Bergstrom, John Reppy, Stephen Rosen, Adam Shaw, Matthew Fluet, and Mike Rainey. Data-Only Flattening for Nested Data Parallelism. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOP)*. 2013.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.

[Ble95] Guy E. Blelloch. *NESL: A Nested Data-Parallel Language*. Tech. rep. Carnegie Mellon University, 1995.

[Ble+94] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a Portable Nested Data-Parallel Language. In *Journal of Parallel and Distributed Computing* 21.1 (1994), pp. 4–14.

[BS89] Guy E. Blelloch and Gary W. Sabot. Compiling Collection-Oriented Languages onto Massively Parallel Computers. In *Journal of Parallel and Distributed Computing* 8.2 (1989), pp. 119–134.

[Bon+06] Peter Boncz, Torsten Grust, Maurice Van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2006, pp. 479–490.

[BK99] Peter Boncz and Martin Kersten. MIL Primitives For Querying a Fragmented World. In *VLDB Journal* 8.2 (1999), pp. 101–119.

[BWK98] Peter Boncz, Annita Wilschut, and Martin Kersten. Flattening an Object Algebra to Provide Performance. In *Proceedings of the Fourteenth International Conference on Data Engineering (ICDE)*. 1998, pp. 568–577.

[BZN05] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Conference on Innovative Data Systems Research (CIDR)*. 2005.

[BBW92] Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally Embedded Query Languages. In *Proceedings of the 4th International Conference on Database Theory (ICDT)*. 1992, pp. 140–154.

[Bru17] Moritz Bruder. MAL Code Generation for Flattening-Based Query Compilers. MA thesis. University of Tübingen, 2017.

[Bun+94] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. In *ACM SIGMOD Record* 23.1 (1994), pp. 87–96.

[Bun+95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsson Wong. Principles of Programming With Complex Objects and Collection Types. In *Theoretical Computer Science* 149.1 (1995), pp. 3–48.

[CB07] Bin Cao and Antonio Badia. SQL Query Optimization Through Nested Relational Algebra. In *ACM Transactions on Database Systems* 32.3 (2007).

[CK00] Manuel M.T. Chakravarty and Gabriele Keller. More Types for Nested Data Parallel Programming. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2000, pp. 94–105.

[CLW13] James Cheney, Sam Lindley, and Philip Wadler. A Practical Theory of Language-Integrated Query. In *ACM SIGPLAN International Conference on Functional Programming*. 2013, pp. 403–416.

[CLW14a] James Cheney, Sam Lindley, and Philip Wadler. Query Shredding: Efficient Relational Evaluation of Queries Over Nested Multisets. In *International Conference on Management of Data (SIGMOD 2014)*. 2014, pp. 1027–1038.

[CLW14b] James Cheney, Sam Lindley, and Philip Wadler. *Query Shredding: Efficient Relational Evaluation of Queries Over Nested Multisets (Extended Version)*. 2014. arXiv: `1404.7078v2 [cs.DB]`.

[Cla+97] Jens Claussen, Alfons Kemper, Guido Moerkotte, and Klaus Peithner. Optimizing Queries with Universal Quantification in Object-Oriented and Object-Relational Databases. In *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB)*. 1997, pp. 286–295.

[CM93] Sophie Cluet and Guido Moerkotte. Nested Queries in Object Bases. In *Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages (DBPL)*. 1993, pp. 226–242.

[Cod72] Edgar F. Codd. Relational Completeness of Data Base Sublanguages. In *Data Base Systems, Proceedings of 6th Courant Computer Science Symposium*. 1972.

[Coo09] Ezra E. Cooper. The Script-Writer's Dream: How to Write Great SQL In Your Own Language, And Be Sure It Will Succeed. In *Database Programming Languages - DBPL 2009, 12th International Symposium*. 2009.

[Coo+06] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects, 5th International Symposium, (FMCO)*. 2006, pp. 266–296.

[CK85] George P. Copeland and Setrag N. Khoshafian. A Decomposition Storage Model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*. 1985, pp. 268–279.

[CM84] George P. Copeland and David Maier. Making Smalltalk a Database System. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. 1984, pp. 316–325.

[TPC-H] Transaction Processing Performance Council. *TPC-H*. URL: `http://www.tpc.org/tpc-h` (visited on 06/16/2018).

[FM00] Leonidas Fegaras and David Maier. Optimizing Object Queries Using an Effective Calculus. In *ACM Transactions on Database Systems* 25.4 (2000), pp. 457–516.

[GW87] Richard A. Ganski and Harry K.T. Wong. Optimization of Nested SQL Queries Revisited. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference (SIGMOD)*. 1987, pp. 23–33.

[Gia17] Paolo G. Giarrusso. Optimizing and Incrementalizing Higher-Order Collection Queries by AST Transformation. PhD thesis. Universität Tübingen, 2017.

[Gia+13] Paolo G. Giarrusso, Klaus Ostermann, Michael Eichberg, Ralf Mitschke, Tillmann Rendel, and Christian Kästner. Reify Your Collection Queries for Modularity and Speed! In *Aspect-Oriented Software Development (AOSD)*. 2013, pp. 1–12.

[Gio+11a] George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. Haskell Boards the Ferry. In *Proceedings of the 22nd International Conference on Implementation and Application of Functional Languages (IFL)*. 2011, pp. 1–18.

[Gio+11b] George Giorgidze, Torsten Grust, Nils Schweinsberg, and Jeroen Weijers. Bringing Back Monad Comprehensions. In *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell*. 2011.

[GRW08] Thomas Goldschmidt, Ralf Reussner, and Jochen Winzen. A Case Study Evaluation of Maintainability and Performance of Persistency Techniques. In *Proceedings of the 13th International Conference on Software Engineering (ICSE)*. 2008, p. 401.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. In *ACM Computing Surveys* 25.2 (1993), pp. 73–170.

[Gru99] Torsten Grust. Comprehending Queries. PhD thesis. Universität Konstanz, 1999.

[Gru02] Torsten Grust. Accelerating XPath Location Steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2002.

[Gru05] Torsten Grust. Purely Relational FLWORs. In *Proceedings of the Second International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*. 2005.

[GM12] Torsten Grust and Manuel Mayr. A Deep Embedding of Queries into Ruby. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering (ICDE)*. 2012, pp. 1257–1260.

[GMR09] Torsten Grust, Manuel Mayr, and Jan Rittinger. XQuery Join Graph Isolation. In *Proceedings of the 25th International Conference on Data Engineering (ICDE)*. 2009.

[GMR10] Torsten Grust, Manuel Mayr, and Jan Rittinger. Let SQL Drive the XQuery Workhorse. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT)*. 2010, pp. 147–158.

[Gru+07] Torsten Grust, Manuel Mayr, Jan Rittinger, Sherif Sakr, and Jens Teubner. A SQL:1999 Code Generator for the Pathfinder XQuery Compiler. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2007, pp. 1162–1164.

[Gru+09] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry Database Supported Program Execution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2009, pp. 1063–1065.

[GR11] Torsten Grust and Jan Rittinger. Observing SQL Queries in their Natural Habitat. In *ACM Transactions on Database Systems* 38.1 (2011).

[GRS10] Torsten Grust, Jan Rittinger, and Tom Schreiber. Avalanche-Safe LINQ Compilation. In *PVLDB* 3.1 (2010), pp. 162–172.

[GST04] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*. 2004, pp. 252–263.

[GS99] Torsten Grust and Marc H. Scholl. How to Comprehend Queries Functionally. In *Journal of Intelligent Information Systems* 12.2-3 (1999), pp. 191–218.

[GU13] Torsten Grust and Alexander Ulrich. First-Class Functions for First-Order Database Engines. In *Proceedings of the 14th International Symposium on Database Programming Languages (DBPL*. 2013.

[GVT03] Torsten Grust, Maurice Van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB)*. 2003.

[HGS16] Jürgen Hölsch, Michael Grossniklaus, and Marc H. Scholl. Optimization of Nested Queries using the NF2 Algebra. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. 2016.

[Hul90] Guy Hulin. On Restructuring Nested Relations in Partitioned Normal Form. In *16th International Conference on Very Large Data Bases (VLDB)*. 1990, pp. 626–637.

[JK84] Matthias Jarke and Jürgen Koch. Query Optimization in Database Systems. In *ACM Computing Surveys* 16.2 (1984), pp. 111–152.

[Kel99] Gabriele Keller. Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines. PhD thesis. Technische Universität Berlin, 1999.

[Kel+12] Gabriele Keller, Manuel M.T. Chakravarty, Ben Lippmeier, and Simon Peyton Jones. Vectorisation Avoidance. In *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell*. 2012, pp. 37–48.

[KS96] Gabriele Keller and Martin Simons. A Calculational Approach to Flattening Nested Data Parallelism in Functional Languages. In *Concurrency and Parallelism, Programming, Networking, and Security: Second Asian Computing Science Conference (ASIAN)*. 1996, pp. 234–243.

[Kim82] Won Kim. On Optimizing an SQL-like Nested Query. In *ACM Transactions on Database Systems* 7.3 (1982), pp. 443–469.

[KLT16] Christoph Koch, 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)*. 2016, pp. 75–90.

[Lei+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How Good Are Query Optimizers, Really. In *PVLDB* 9.3 (2015), pp. 204–215.

[LS03] Alberto Lerner and Dennis Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *Proceedings of the 29th VLDB Conference*. 2003, pp. 345–356.

[Les05] Roman Leshchinskiy. Higher-Order Nested Data Parallelism: Semantics and Implementation. PhD thesis. Technische Universität Berlin, 2005.

[LS97] Alon Levy and Dan Suciu. Deciding Containment for Queries with Complex Objects and Aggregations. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 1997, pp. 20–31.

[LW97] Leonid Libkin and Limsoon Wong. Query Languages for Bags and Aggregate Functions. In *Journal of Computer and System Sciences* 55.2 (1997), pp. 241–272.

[Lip+12] Ben Lippmeier, Manuel M.T. Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Simon Peyton Jones. Work Efficient Higher-Order Vectorisation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2012, pp. 259–270.

[Mad16] Frederik M. Madsen. Streaming for Functional Data-Parallel Languages. PhD thesis. University of Copenhagen, 2016.

[MKB09] Stefan Manegold, Martin Kersten, and Peter Boncz. Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct. In *PVLDB* 2.2 (2009), pp. 1648–1653.

[Mar10] Simon Marlow. *Haskell 2010: Language Report*. `haskell.org`. 2010.

[MHM04] Norman May, Sven Helmer, and Guido Moerkotte. Nested Queries and Quantifiers in an Ordered Context. In *Proceedings of the 20th International Conference on Data Engineering (ICDE)*. 2004, pp. 239–250.

[May13] Manuel Mayr. A Deep Embedding of Queries into Ruby. PhD thesis. Universität Tübingen, 2013.

[MBB06] Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2006, p. 706.

[MN11] Guido Moerkotte and Thomas Neumann. Accelerating Queries with Group-By and Join by Groupjoin. In *PVLDB* 4.11 (2011), pp. 843–851. ISSN: 21508097.

[Nag15] Fabian Nagel. Efficient Query Processing in Managed Runtimes. PhD thesis. University of Edinburgh, 2015.

[NBV14] Fabian Nagel, Gavin Bierman, and Stratis D. Viglas. Code Generation for Efficient Query Processing in Managed Runtimes. In *PVLDB* 7.12 (2014), pp. 1095–1106.

[Nak90] Ryohey Nakano. Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions. In *ACM Transactions on Database Systems* 15.4 (1990), pp. 518–557.

[Neu11] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. In *PVLDB* 4.9 (2011), pp. 539–550.

[Pal+17] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. *Weld: Rethinking the Interface Between Data-Intensive Applications*. 2017. arXiv: 1709.06416 [cs.DB].

[PP95] Daniel W. Palmer and Jan F. Prins. Work-Efficient Nested Data-Parallelism. In *Frontiers of Massively Parallel Computation*. 1995.

[Pal+95] Daniel W. Palmer, Jan F. Prins, Siddhartha Chatterjee, and Rickard Faith. Piecewise Execution of Nested Data-Parallel Programs. In *Languages and Compilers for Parallel Computing, 8th International Workshop (LCPC)*. 1995, pp. 346–361.

[PV92] Jan Paredaens and Jan Van Gucht. Converting Nested Algebra Expressions into Flat Algebra Expressions. In *ACM Transactions on Database Systems* 17.1 (1992), pp. 65–93.

[PW07] Simon L. Peyton Jones and Philip Wadler. Comprehensive Comprehensions. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*. 2007, pp. 61–72.

[Pey+08] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M.T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Foundations of Software Technology and Theoretical Computer Science*. 2008, pp. 383–414.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[Pir+16] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. In *PVLDB* 9.14 (2016), pp. 1707–1718.

[PP93] Jan F. Prins and Daniel W. Palmer. Transforming High-Level Data-Parallel Programs into Vector Operations. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*. Vol. 3175. 1993.

[RP00] James Riely and Jan F. Prins. Flattening is an Improvement. In *7th International Symposium on Static Analysis*. 2000, pp. 1–17.

[Rit11] Jan Rittinger. Constructing a Relational Query Optimizer for Non-Relational Languages. PhD thesis. Universität Tübingen, 2011.

[Rom12] Tiark Rompf. Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level Performance Programming. PhD thesis. Ecole Polytechnique Federale de Lausanne, 2012.

[RKS88] Mark A. Roth, Henry F. Korth, and Abraham Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. In *ACM Transactions on Database Systems* 13.4 (1988), pp. 389–417.

[SFG14] Neil Sculthorpe, Nicolas Frisby, and Andy Gill. The Kansas University Rewrite Engine - A Haskell-Embedded Strategic Programming Language with Custom Closed Universes. In *Journal of Functional Programming* 24.4 (2014), pp. 434–473.

[Sha+16] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. 2016, pp. 1907–1922.

[Shu+12] Jeff Shute et al. F1: A Distributed SQL Database That Scales. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2012.

[SB90] Jay M. Sipelstein and Guy E. Blelloch. *Collection-Oriented Languages*. Tech. rep. Carnegy Mellon University, 1990.

[Slick] *Slick: Functional Relational Mapping for Scala*. URL: `http://slick.lightbend.com` (visited on 06/16/2018).

[Ste95] Hennie J. Steenhagen. Optimization of Object Query Languages. PhD thesis. Universiteit Twente, 1995.

[SAB94] Hennie J. Steenhagen, Peter M.G. Apers, and Henk M. Blanken. Optimization of Nested Queries in a Complex Object Model. In *Proceedings of the 4th International Conference on Extending Database Technology*. 1994, pp. 337–350.

[Ste+94] Hennie J. Steenhagen, Peter M.G. Apers, Henk M. Blanken, and Rolf A. de By. From Nested-Loop to Join Queries in OODB. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB)*. 1994, pp. 618–629.

[SC18] Jan Stolarek and James Cheney. Language-Integrated Provenance in Haskell. In *Programming Journal* 2.3 (2018), p. 11.

[Suc95] Dan Suciu. Parallel Programming Languages For Collections. PhD. University of Pennsylvania, 1995.

[Suc96] Dan Suciu. Implementation and Analysis of a Parallel Collection Query Language. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB)*. 1996, pp. 366–377.

[Suc97] Dan Suciu. Bounded Fixpoints for Complex Objects. In *Theoretical Computer Science* 176.1-2 (1997), pp. 283–328.

[ST94] Dan Suciu and Val Tannen. Efficient Compilation of High-Level Data Parallel Algorithms. In *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 1994, pp. 57–66.

[PG] *The PostgreSQL Relational Database System*. URL: `http://www.postgresql.org` (visited on 06/16/2018).

[Tri91] Philip W. Trinder. Comprehensions, a Query Notation for DBPLs. In *Proceedings of the Third International Workshop on Database Programming Languages (DBPL)*. 1991.

[Ulr11] Alexander Ulrich. A Ferry-Based Query Backend for the Links Programming Language. MA thesis. University of Tübingen, 2011.

[UG15] Alexander Ulrich and Torsten Grust. The Flatter, the Better. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2015, pp. 1421–1426.

[Van99] Jan Van Den Bussche. Simulation of the Nested Relational Algebra by the Flat Relational Algebra, with an Application to the Complexity of Evaluating Powerset Algebra Expressions. In *Theoretical Computer Science* 254.1 (1999), pp. 363–377.

[Van06] Joeri Van Ruth. Flattening Queries over Nested Data Types. PhD thesis. University of Twente, 2006.

[Wad90] Philip Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. 1990, pp. 61–78.

[Wei17] Jeroen Weijers. A Flattening Based Compilation Strategy For Rich Database Query Languages. 2017.

[Won94] Limsoon Wong. Querying Nested Collections. PhD thesis. University of Pennsylvania, 1994.

[Won96] Limsoon Wong. Normal Forms and Conservative Extension Properties for Query Languages over Collection Types. In *Journal of Computer and System Sciences* 52.3 (1996), pp. 495–505.

[Won00] Limsoon Wong. Kleisli, a Functional Query System. In *Journal of Functional Programming* 10.1 (2000), pp. 19–56.

[Yan+17] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *Proceedings of the 2017 ACM Conference on Information and Knowledge Management (CIKM)*. 2017, pp. 1299–1308.