# Query Shredding: Efficient Relational Evaluation of Queries over Nested Multisets

James Cheney
University of Edinburgh
jcheney@inf.ed.ac.uk

Sam Lindley
University of Edinburgh
Sam.Lindley@ed.ac.uk

Philip Wadler
University of Edinburgh
wadler@inf.ed.ac.uk

## ABSTRACT

Nested relational query languages have been explored extensively, and underlie industrial language-integrated query systems such as Microsoft's LINQ. However, relational databases do not natively support nested collections in query results. This can lead to major performance problems: if programmers write queries that yield nested results, then such systems typically either fail or generate a large number of queries. We present a new approach to query shredding, which converts a query returning nested data to a fixed number of SQL queries. Our approach, in contrast to prior work, handles multiset semantics, and generates an idiomatic SQL:1999 query directly from a normal form for nested queries. We provide a detailed description of our translation and present experiments showing that it offers comparable or better performance than a recent alternative approach on a range of examples.

## Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

## Keywords

language-integrated query; querying nested collections

## 1. INTRODUCTION

Databases are one of the most important applications of declarative programming techniques. However, relational databases only support queries against flat tables, while programming languages typically provide complex data structures that allow arbitrary combinations of types including nesting of collections (e.g. sets of sets). Motivated by this so-called *impedance mismatch*, and inspired by insights into language design based on monadic comprehensions [28], database researchers introduced nested relational query languages [22, 3, 4] as a generalisation of flat relational queries to allow nesting collection types inside records or other types. Several recent language designs, such as XQuery [21] and PigLatin [20], have further extended these ideas, and they have been particularly influential on language-integrated querying systems such as Kleisli [30], Microsoft's LINQ in C# and F# [19, 24, 5], Links [8, 18], and Ferry [10].

*SIGMOD'14,* June 22–27, 2014, Snowbird, UT, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2612186.

This paper considers the problem of translating nested queries over nested data to flat queries over a flat representation of nested data, or *query shredding* for short. Our motivation is to support a free combination of the features of nested relational query languages with those of high-level programming languages, particularly systems such as Links, Ferry, and LINQ. All three of these systems support queries over nested data structures (e.g. records containing nested sets, multisets/bags, or lists) in principle; however, only Ferry supports them in practice. Links and LINQ currently either reject such queries at run-time or execute them inefficiently in-memory by loading unnecessarily large amounts of data or issuing large numbers of queries (sometimes called *query storms* or *avalanches* [11] or the $N + 1$ *query problem*). To construct nested data structures while avoiding this performance penalty, programmers must currently write flat queries (e.g. loading in a superset of the needed source data) and convert the results to nested data structures. Manually reconstructing nested query results is tricky and hard to maintain; it may also mask optimisation opportunities.

In the Ferry system, Grust et al. [10, 11] have implemented shredding for nested list queries by adapting an XQuery-to-SQL translation called *loop-lifting* [14]. Loop-lifting produces queries that make heavy use of advanced On-Line Analytic Processing (OLAP) features of SQL:1999, such as `ROW_NUMBER` and `DENSE_RANK`, and to optimise these queries Ferry relies on a SQL:1999 query optimiser called Pathfinder [13].

Van den Bussche [26] proved expressiveness results showing that it is possible in principle to evaluate nested queries over *sets* via multiple flat queries. To strengthen the result, Van den Bussche's simulation eschews value invention mechanisms such as SQL:1999's `ROW_NUMBER`. The downside, however, is that the flat queries can produce results that are quadratically larger than needed to represent sets and may not preserve bag semantics.

Query shredding is related to the well-studied *query unnesting* problem [16, 9]. However, most prior work on unnesting only considers SQL queries that contain subqueries in WHERE clauses, not queries returning nested results; the main exception is Fegaras and Maier's work on query unnesting in a complex object calculus [9].

In this paper, we introduce a new approach to query shredding for nested multiset queries (a case not handled by prior work). Our work is formulated in terms of the Links language, but should be applicable to other language-integrated query systems, such as Ferry and LINQ, or to other complex-object query languages [9]. Figure 1 illustrates the behaviour of Links, Ferry and our approach.

We decompose the translation from nested to flat queries into a series of simpler translations. We leverage prior work on *normalisation* that translates a higher-order query to a *normal form* in

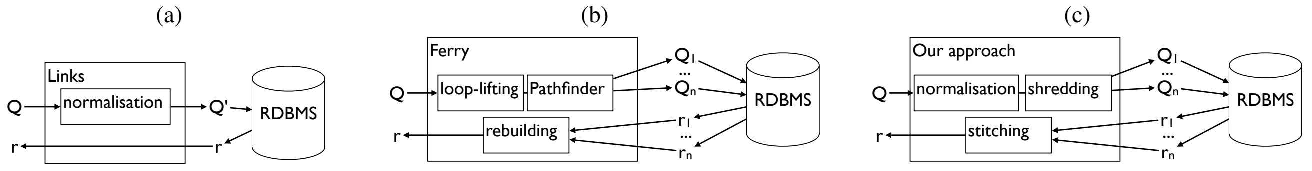

**Figure 1: (a) Default Links behaviour (flat queries) (b) Ferry (nested list queries) (c) Our approach (nested bag queries)**

which higher-order features have been eliminated [29, 7, 9]. Our algorithm operates on normal forms. We review normalisation in Section 2, and give a running example of our approach in Section 3.

Sections 4–7 present the remaining phases of our approach, which are new. The *shredding* phase translates a single, nested query to a number of flat queries. These queries are organised in a *shredded package*, which is essentially a type expression whose collection type constructors are annotated with queries. The different queries are linked by *indexes*, that is, keys and foreign keys. The shredding translation is presented in Section 4. Shredding leverages the normalisation phase in that we can define translations on types and terms independently. Section 5 shows how to run shredded queries and stitch the results back together to form nested values.

The *let-insertion* phase implements a flat indexing scheme using a let-binding construct and a row-numbering operation. Let-insertion (described in Section 6) is conceptually straightforward, but provides a vital link to proper SQL by providing an implementation of abstract indexes. The final stage is to translate to SQL (Section 7) by flattening records, translating let-binding to SQL's `WITH`, and translating row-numbering to SQL's `ROW_NUMBER`.

We have implemented and experimentally evaluated our approach (Section 8) in comparison with Ulrich's implementation of loop-lifting in Links [25]. Our approach typically performs as well or better than loop-lifting, and can be significantly faster.

Our contribution over prior work can be summarised as follows. Fegaras and Maier [9] show how to unnest complex object queries including lists, sets, and bags, but target a nonstandard nested relational algebra, whereas we target standard SQL. Van den Bussche's simulation [26] translates nested set queries to several relational queries but was used to prove a theoretical result and was not intended as a practical implementation technique, nor has it been implemented and evaluated. Ferry [10, 11] translates nested list queries to several SQL:1999 queries and then tries to simplify the resulting queries using Pathfinder. Sometimes, however, this produces queries with cross-products inside OLAP operations such as `ROW_NUMBER`, which Pathfinder cannot simplify. In contrast, we delay introducing OLAP operations until the last stage, and our experiments show how this leads to much better performance on some queries. Finally, we handle nested multisets, not sets or lists.

Additional details, proofs of correctness, and comparison with related work are available in a companion technical report [6].

## 2. BACKGROUND

We use metavariables $x, y, \ldots, f, g$ for *variables*, and $c, d, \ldots$ for *constants* and primitive operations. We also use letters $t, t', \ldots$ for *table names*, $\ell, \ell', \ell_i, \ldots$ for *record labels* and $a, b, \ldots$ for *tags*.

We write $M[x := N]$ for capture-avoiding substitution of $N$ for $x$ in $M$. We write $\vec{x}$ for a vector $x_1, \ldots, x_n$. Moreover, we extend vector notation pointwise to other constructs, writing, for example, $\langle\overrightarrow{\ell = M}\rangle$ for $\langle \ell_1 = M_1, \ldots, \ell_n = M_n\rangle$.

We write: square brackets $[-]$ for the meta level list constructor; $w :: \vec{v}$ for adding the element $w$ onto the front of the list $\vec{v}$; $\vec{v} +\!\!+ \vec{w}$ for the result of appending the list $\vec{w}$ onto the end of the list $\vec{v}$; and

$$
\begin{aligned}
\mathcal{N}[\![x]\!]_\rho &= \rho(x) \\
\mathcal{N}[\![c(X_1, \ldots, X_n)]\!]_\rho &= [\![c]\!](\mathcal{N}[\![X_1]\!]_\rho, \ldots, \mathcal{N}[\![X_n]\!]_\rho) \\
\mathcal{N}[\![\lambda x.M]\!]_\rho &= \lambda v.\mathcal{N}[\![M]\!]_{\rho[x\mapsto v]} \\
\mathcal{N}[\![M\;N]\!]_\rho &= \mathcal{N}[\![M]\!]_\rho(\mathcal{N}[\![N]\!]_\rho) \\
\mathcal{N}[\![\langle \ell_i = M_i\rangle_{i=1}^n]\!]_\rho &= \langle \ell_i = \mathcal{N}[\![M_i]\!]_\rho\rangle_{i=1}^n \\
\mathcal{N}[\![M.\ell]\!]_\rho &= \mathcal{N}[\![M]\!]_\rho.\ell \\
\mathcal{N}[\![\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N]\!]_\rho &= \begin{cases} \mathcal{N}[\![M]\!]_\rho, & \text{if } \mathcal{N}[\![L]\!]_\rho = \mathsf{true} \\ \mathcal{N}[\![N]\!]_\rho, & \text{if } \mathcal{N}[\![L]\!]_\rho = \mathsf{false} \end{cases} \\
\mathcal{N}[\![\mathsf{return}\ M]\!]_\rho &= [\mathcal{N}[\![M]\!]_\rho] \\
\mathcal{N}[\![\emptyset]\!]_\rho &= [\,] \\
\mathcal{N}[\![M \uplus N]\!]_\rho &= \mathcal{N}[\![M]\!]_\rho +\!\!+ \mathcal{N}[\![N]\!]_\rho \\
\mathcal{N}[\![\mathsf{for}\ (x \leftarrow M)\ N]\!]_\rho &= \mathit{concat}[\mathcal{N}[\![N]\!]_{\rho[x\mapsto v]} \mid v \leftarrow \mathcal{N}[\![M]\!]_\rho] \\
\mathcal{N}[\![\mathsf{empty}\ M]\!]_\rho &= \begin{cases} \mathsf{true}, & \text{if } \mathcal{N}[\![M]\!]_\rho = [\,] \\ \mathsf{false}, & \text{if } \mathcal{N}[\![M]\!]_\rho \neq [\,] \end{cases} \\
\mathcal{N}[\![\mathsf{table}\ t]\!]_\rho &= [\![t]\!]
\end{aligned}
$$

**Figure 2: Semantics of $\lambda_{NRC}$**

*concat* for the function that concatenates a list of lists. We also make use of the following functions:

$$
\mathit{init}\ [x_i]_{i=1}^n = [x_i]_{i=1}^{n-1} \qquad \mathit{last}\ [x_i]_{i=1}^n = x_n
$$
$$
\mathit{enum}([v_1, \ldots, v_m]) = [\langle 1, v_1\rangle, \ldots \langle m, v_m\rangle]
$$

In the meta language we make extensive use of comprehensions, primarily list comprehensions. For instance, $[v \mid x \leftarrow xs, y \leftarrow ys, p]$, returns a copy of $v$ for each pair $\langle x, y\rangle$ of elements of $xs$ and $ys$ such that the predicate $p$ holds. We write $[v_i]_{i=1}^n$ as shorthand for $[v_i \mid 1 \leq i \leq n]$ and similarly, e.g., $\langle \ell_i = M_i\rangle_{i=1}^n$ for $\langle \ell_1 = M_1, \ldots, \ell_n = M_n\rangle$.

### 2.1 Nested relational calculus

We take the higher-order, nested relational calculus (evaluated over bags) as our starting point. We call this $\lambda_{NRC}$; this is essentially a core language for the query components of Links, Ferry, and LINQ. The types of $\lambda_{NRC}$ include base types (integers, strings, booleans), record types $\langle\overrightarrow{\ell : A}\rangle$, bag types $\mathrm{Bag}\,A$, and function types $A \to B$.

$$
\begin{array}{ll}
\text{Types} & A, B ::= O \mid \langle\overrightarrow{\ell : A}\rangle \mid \mathrm{Bag}\,A \mid A \to B \\
\text{Base types} & O ::= \mathit{Int} \mid \mathit{Bool} \mid \mathit{String}
\end{array}
$$

We say that a type is *nested* if it contains no function types and *flat* if it contains only base and record types.

The terms of $\lambda_{NRC}$ include $\lambda$-abstractions, applications, and the standard terms of nested relational calculus.

$$
\begin{array}{lll}
\text{Terms} & M, N ::= & x \mid c(\vec{M}) \mid \mathsf{table}\ t \mid \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ N' \\
& \mid & \lambda x.\, M \mid M\; N \mid \langle\overrightarrow{\ell = M}\rangle \mid M.\ell \mid \mathsf{empty}\ M \\
& \mid & \mathsf{return}\ M \mid \emptyset \mid M \uplus N \mid \mathsf{for}\ (x \leftarrow M)\ N
\end{array}
$$

We assume that the constants and primitive functions include boolean values with negation and conjunction, and integer values with standard arithmetic operations and equality tests. We assume special labels $\#_1, \#_2, \ldots$ and encode tuple types $\langle A_1, \ldots, A_n\rangle$

as record types $\langle \#_1 : A_1, \ldots, \#_n : A_n \rangle$, and similarly tuple terms $\langle M_1, \ldots, M_n \rangle$ as record terms $\langle \#_1 = M_1, \ldots, \#_n = M_n \rangle$. We assume fixed signatures $\Sigma(t)$ and $\Sigma(c)$ for tables and constants. The tables are constrained to have flat relation type $(\mathrm{Bag}\,\langle \ell_1 : O_1, \ldots, \ell_n : O_n \rangle)$, and the constants must be of base type or first order $n$-ary functions $(\langle O_1, \ldots, O_n \rangle \to O)$.

Most language constructs are standard. The $\emptyset$ expression builds an empty bag, return $M$ constructs a singleton, and $M \uplus N$ builds the bag union of two collections. The for $(x \leftarrow M)\ N$ comprehension construct iterates over a bag obtained by evaluating $M$, binds $x$ to each element, evaluates $N$ to another bag for each such binding, and takes the union of the results. The expression empty $M$ evaluates to true if $M$ evaluates to an empty bag, false otherwise.

$\lambda_{NRC}$ employs a standard type system similar to that presented in other work [30, 18, 5]. We will also employ several typed intermediate languages and translations mapping $\lambda_{NRC}$ to SQL. All of these (straightforward) type systems are omitted due to space limits; they will be available in the full version of this paper.

*Semantics.* We give a denotational semantics in terms of lists. Though we wish to preserve bag semantics, we interpret object-level bags as meta-level lists. For meta-level values $v$ and $v'$, we consider $v$ and $v'$ equivalent as multisets if they are equal up to permutation of list elements. We use lists mainly so that we can talk about order-sensitive operations such as row_number.

We interpret base types as integers, booleans and strings, function types as functions, record types as records, and bag types as lists. For each table $t \in dom(\Sigma)$, we assume a fixed interpretation $[\![t]\!]$ of $t$ as a list of records of type $\Sigma(t)$. In SQL, tables do not have a list semantics by default, but we can impose one by choosing a canonical ordering for the rows of the table. We order by all of the columns arranged in lexicographic order (assuming linear orderings on field names and base types).

We assume fixed interpretations $[\![c]\!]$ for the constants and primitive operations. The semantics of nested relational calculus are shown in Figure 2. We let $\rho$ range over environments mapping variables to values, writing $\varepsilon$ for the empty environment and $\rho[x \mapsto v]$ for the extension of $\rho$ with $x$ bound to $v$.

## 2.2 Query normalisation

In Links, query normalisation is an important part of the execution model [7, 18]. Links currently supports only queries mapping flat tables to flat results, or *flat–flat* queries. When a subexpression denoting a query is evaluated, the subexpression is first normalised and then converted to SQL, which is sent to the database for evaluation; the tuples received in response are then converted into a Links value and normal execution resumes (see Figure1(a)).

For *flat–nested* queries that read from flat tables and produce a nested result value, our normalisation procedure is similar to the one currently used in Links [18], but we hoist all conditionals into the nearest enclosing comprehension as where clauses. This is a minor change; the modified algorithm is given in the full version of this paper. The resulting normal forms are:

| | |
|---|---|
| Query terms | $L ::= \biguplus \vec{C}$ |
| Comprehensions | $C ::= \mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}\ M$ |
| Generators | $G ::= x \leftarrow t$ |
| Normalised terms | $M, N ::= X \mid R \mid L$ |
| Record terms | $R ::= \langle \overrightarrow{\ell = M} \rangle$ |
| Base terms | $X ::= x.\ell \mid c(\vec{X}) \mid \mathsf{empty}\ L$ |

Any closed flat–nested query can be converted to an equivalent term in the above normal form.

$[\![departments]\!] =$

| (id) | name |
|---|---|
| 1 | Product |
| 2 | Quality |
| 3 | Research |
| 4 | Sales |

$[\![employees]\!] =$

| (id) | dept | name | salary |
|---|---|---|---|
| 1 | Product | Alex | 20000 |
| 2 | Product | Bert | 900 |
| 3 | Research | Cora | 50000 |
| 4 | Research | Drew | 60000 |
| 5 | Sales | Erik | 2000000 |
| 6 | Sales | Fred | 700 |
| 7 | Sales | Gina | 100000 |

$[\![tasks]\!] =$

| (id) | employee | task |
|---|---|---|
| 1 | Alex | build |
| 2 | Bert | build |
| 3 | Cora | abstract |
| 4 | Cora | build |
| 5 | Cora | call |
| 6 | Cora | dissemble |
| 7 | Cora | enthuse |
| 8 | Drew | abstract |
| 9 | Drew | enthuse |
| 10 | Erik | call |
| 11 | Erik | enthuse |
| 12 | Fred | call |
| 13 | Gina | call |
| 14 | Gina | dissemble |

$[\![contacts]\!] =$

| (id) | dept | name | client |
|---|---|---|---|
| 1 | Product | Pam | false |
| 2 | Product | Pat | true |
| 3 | Research | Rob | false |
| 4 | Research | Roy | false |
| 5 | Sales | Sam | false |
| 6 | Sales | Sid | false |
| 7 | Sales | Sue | true |

**Figure 3: Sample data**

THEOREM 1. *Given a closed flat–nested query* $\vdash M : \mathrm{Bag}\ A$, *there exists a normalisation function* $norm_{\mathrm{Bag}\ A}$, *mapping each* $M$ *to an equivalent normal form* $norm_{\mathrm{Bag}\ A}(M)$.

The normalisation algorithm and correctness proof are similar to those in previous papers [7, 18, 5]. The normal forms above can also be viewed as an SQL-like language allowing relation-valued attributes (similar to the complex-object calculus of Fegaras and Maier [9]). Thus, our results can also be used to support nested query results in an SQL-like query language. In this paper, however, we focus on the functional core language based on comprehensions, as illustrated in the next section.

## 3. RUNNING EXAMPLE

To motivate and illustrate our work, we present an extended example showing how our shredding translation could be used to provide useful functionality to programmers working in LINQ using F#, Ferry or Links. We first describe the code the programmer would actually write and the results the system produces. Throughout the rest of the paper, we return to this example to illustrate how the shredding translation works.

Consider a flat database schema $\Sigma$ for an organisation:

$$\begin{array}{l} tasks(\mathsf{employee} : String, \mathsf{task} : String) \\ employees(\mathsf{dept} : String, \mathsf{name} : String, \mathsf{salary} : Int) \\ contacts(\mathsf{dept} : String, \mathsf{name} : String, \mathsf{client} : Bool) \\ departments(\mathsf{name} : String) \end{array}$$

Each department has a name, a collection of employees, and a collection of external contacts. Each employee has a name, salary and a collection of tasks. Some contacts are clients. Figure 3 shows a small instance of this schema. For convenience, we also assume every table has an integer-valued key *id*.

In $\lambda_{NRC}$, queries of the form for … where … return … are natively supported. These are *comprehensions* as found in XQuery or functional programming languages, and they generalise idiomatic SQL `SELECT FROM WHERE` queries [3]. Unlike SQL, we can use (nonrecursive) functions to define queries with parameters, or parts of queries, and freely combine them. For example, the following functions define useful query patterns over the above schema:

```
taskOfEmp e = for (t ← tasks)
                where (t.employee = e.name)
                return t.tsk
contactsOfDept d = for (c ← contacts)
                     where (d.dept = c.dept)
                     return ⟨name = c.name, client = c.client⟩
employeesByTask t =
    for (e ← employees, d ← departments)
    where (e.name = t.employee ∧ e.dept = d.dept)
    return ⟨b = e.employee, c = d.dept⟩
```

Nested queries allow free mixing of collection (bag) types with record or base types. For example, the following query

```
employeesOfDept d = for (e ← employees)
                      where (d.dept = e.dept)
                      return ⟨name = e.name, salary = e.salary,
                              tasks = tasksOfEmp e⟩
```

returns a nested result: a collection of employees in a department, each with an associated collection of tasks. That is, its return type is Bag $\langle$name:$String$, salary:$Int$, tasks:Bag $String\rangle$

Consider the following nested schema for organisations:

$$\begin{aligned}
Task &= String\\
Employee &= \langle \text{name} : String, \text{salary} : Int, \text{tasks} : \text{Bag}\ Task\rangle\\
Contact &= \langle \text{name} : String, \text{client} : Bool\rangle\\
Department &= \langle \text{name} : String, \text{employees} : \text{Bag}\ Employee,\\
&\qquad \text{contacts} : \text{Bag}\ Contact\rangle\\
Organisation &= \text{Bag}\ Department
\end{aligned}$$

Using some of the above functions we can write a query $Q_{org}$ that maps data in the flat schema $\Sigma$ to the nested type $Organisation$, as follows:

```
Q_org = for (d ← departments)
          return (⟨name = d.name,
                   employees = employeesOfDept d,
                   contacts = contactsOfDept d⟩
```

We can also define and use higher-order functions to build queries, such as the following:

```
filter p xs = for (x ← xs) where (p(x)) return x
any xs p = ¬(empty(for (x ← xs) where (p(x)) return ⟨⟩))
all xs p = ¬(any xs (λx.¬(p(x))))
contains xs u = any xs (λx.x = u)
```

To illustrate the main technical challenges of shredding, we consider a query with two levels of nesting and a union operation.

Suppose we wish to find for each department a collection of people of interest, both employees and contacts, along with a list of the tasks they perform. Specifically, we are interested in those employees that earn less than a thousand euros and those who earn more than a million euros, call them *outliers*, along with those contacts who are clients. The following code defines poor, rich, outliers, and clients:

```
isPoor x = x.salary < 1000
isRich x = x.salary > 1000000
outliers xs = filter (λx.isRich x ∨ isPoor x) xs
clients xs = filter (λx. x.client) xs
```

We also introduce a convenient higher-order function that uses its $f$ parameter to initialise the tasks of its elements:

```
getTasks xs f = for (x ← xs) return ⟨name = x.name, tasks = f x⟩
```

Using the above operations, the query $Q$ returns each department, the outliers and clients associated with that department, and their tasks. We assign the special task "buy" to clients.

```
Q(organisation) =
   for (x ← organisation)
   return (⟨department = x.name,
            people =
              getTasks(outliers(x.employees)) (λy. y.tasks)
            ⊎ getTasks(clients(x.contacts)) (λy. return "buy")⟩)
```

The result type of $Q$ is:

$$\begin{aligned}
Result = \text{Bag}\ \langle &\text{department} : String,\\
&\text{people} : \text{Bag}\ \langle \text{name} : String, \text{tasks} : \text{Bag}\ String\rangle\rangle
\end{aligned}$$

We can compose $Q$ with $Q_{org}$ to form a query $Q(Q_{org})$ from the flat data stored in $\Sigma$ to the nested $Result$. The normal form of this composed query, which we call $Q_{comp}$, is as follows:

```
Q_comp = for (x ← departments)
          return (
           ⟨department = x.name,
            people =
              (for (y ← employees) where (x.name = y.dept ∧
                     (y.salary < 1000 ∨ y.salary > 1000000))
               return (⟨name = y.name,
                         tasks = for (z ← tasks)
                                   where (z.employee = y.name)
                                   return z.task⟩))
            ⊎ (for (y ← contacts)
               where (x.name = y.dept ∧ y.client)
               return (⟨name = y.name,
                         tasks = return "buy"⟩))⟩))
```

The result of running $Q_{comp}$ on the data in Figure 3 is:

```
[⟨department = "Product",
  people = [⟨name = "Bert", tasks = ["build"]⟩,
            ⟨name = "Pat", tasks = ["buy"]⟩]⟩,
 ⟨department = "Research", people = ∅⟩,
 ⟨department = "Quality", people = ∅⟩,
 ⟨department = "Sales",
  people = [⟨name = "Erik", tasks = ["call", "enthuse"]⟩,
            ⟨name = "Fred", tasks = ["call"]⟩,
            ⟨name = "Sue", tasks = ["buy"]⟩]⟩]
```

Now, however, we are faced with a problem: SQL databases do not directly support nested multisets (or sets). Our shredding translation, like Van den Bussche's simulation for sets [26] and Grust et al.'s for lists [11], can translate a normalised query such as $Q_{comp}$ : $Result$ that maps flat input $\Sigma$ to nested output $Result$ to a fixed number of flat queries $q_1 : Result_1, \ldots, q_n : Result_n$ whose results can be combined via a *stitching* operation $Q_{stitch} : Result_1 \times \cdots \times Result_n \rightarrow Result$. Thus, we can simulate the query $Q_{comp}$ by running $q_1, \ldots, q_n$ remotely on the database and stitching the results together using $Q_{stitch}$. The number of intermediate queries $n$ is the *nesting degree* of $Result$, that is, the number of collection type constructors in the result type. For example, the nesting degree of Bag $\langle A$ : Bag $Int, B$ : Bag $String\rangle$ is 3. The nesting degree of the type $Result$ is also 3, which means $Q$ can be shredded into three flat queries.

The basic idea is straightforward. Whenever a nested bag appears in the output of a query, we generate an index that uniquely identifies the current context. Then a separate query produces the contents of the nested bag, where each element is paired up with its parent index. Each inner level of nesting requires a further query.

We will illustrate by showing the results of the three queries and how they can be combined to reconstitute the desired nested result.

The outer query $q_1$ contains one entry for each department, with an index $\langle a, id\rangle$ in place of each nested collection:

$$\begin{aligned} r_1 = [&\langle \mathsf{department} = \text{“Product”}, \mathsf{people} = \langle a, 1\rangle\rangle,\\ &\langle \mathsf{department} = \text{“Quality”}, \mathsf{people} = \langle a, 2\rangle\rangle,\\ &\langle \mathsf{department} = \text{“Research”}, \mathsf{people} = \langle a, 3\rangle\rangle,\\ &\langle \mathsf{department} = \text{“Sales”}, \mathsf{people} = \langle a, 4\rangle\rangle] \end{aligned}$$

The second query $q_2$ generates the data needed for the people collections:

$$\begin{aligned} r_2 = [&\langle\langle a, 1\rangle, \langle \mathsf{name} = \text{“Bert”}, \mathsf{tasks} = \langle b, 1, 2\rangle\rangle\rangle,\\ &\langle\langle a, 4\rangle, \langle \mathsf{name} = \text{“Erik”}, \mathsf{tasks} = \langle b, 4, 5\rangle\rangle\rangle,\\ &\langle\langle a, 4\rangle, \langle \mathsf{name} = \text{“Fred”}, \mathsf{tasks} = \langle b, 4, 6\rangle\rangle\rangle,\\ &\langle\langle a, 1\rangle, \langle \mathsf{name} = \text{“Pat”}, \mathsf{tasks} = \langle d, 1, 2\rangle\rangle\rangle,\\ &\langle\langle a, 4\rangle, \langle \mathsf{name} = \text{“Sue”}, \mathsf{tasks} = \langle d, 4, 7\rangle\rangle\rangle] \end{aligned}$$

The idea is to ensure that we can stitch the results of $q_1$ together with the results of $q_2$ by joining the *inner indexes* of $q_1$ (bound to the people field of each result) with the *outer indexes* of $q_2$ (bound to the first component of each result). In both cases the static components of these indexes are the same tag $a$. Joining the people field of $q_1$ to the outer index of $q_2$ correctly associates each person with the appropriate department.

Finally, let us consider the results of the innermost query $q_3$ for generating the bag bound to the tasks field:

$$\begin{aligned} r_3 = [&\langle\langle b, 1, 2\rangle, \text{“build”}\rangle, \langle\langle b, 4, 5\rangle, \text{“call”}\rangle, \langle\langle b, 4, 5\rangle, \text{“enthuse”}\rangle,\\ &\langle\langle b, 4, 6\rangle, \text{“call”}\rangle, \langle\langle d, 1, 2\rangle, \text{“buy”}\rangle, \langle\langle d, 4, 7\rangle, \text{“buy”}\rangle] \end{aligned}$$

Recall that $q_2$ returns further inner indexes for the tasks associated with each person. The two halves of the union have different static indexes for the tasks $b$ and $d$, because they arise from different comprehensions in the source term. Furthermore, the dynamic index now consists of two id fields ($x$.id and $y$.id) in each half of the union. Thus, joining the tasks field of $q_2$ to the outer index of $q_3$ correctly associates each task with the appropriate outlier.

Note that each of the queries $q_1, q_2, q_3$ produces records that contain other records as fields. This is not strictly allowed by SQL, but it is straightforward to simulate such nested records by rewriting to a query with no nested collections in its result type; this is similar to Van den Bussche's simulation [26]. However, this approach incurs extra storage and query-processing cost. Later in the paper, we explore an alternative approach which collapses the indexes at each level to a pair $\langle a, i\rangle$ of static index and a single “surrogate” integer, similarly to Ferry's approach [11]. For example, using this approach we could represent the results of $q_2$ and $q_3$ as follows:

$$\begin{aligned} r_2' = [&\langle\langle a, 1\rangle, \langle \mathsf{name} = \text{“Bert”}, \mathsf{tasks} = \langle b, 1\rangle\rangle\rangle,\\ &\langle\langle a, 4\rangle, \langle \mathsf{name} = \text{“Erik”}, \mathsf{tasks} = \langle b, 2\rangle\rangle\rangle,\\ &\langle\langle a, 4\rangle, \langle \mathsf{name} = \text{“Fred”}, \mathsf{tasks} = \langle b, 3\rangle\rangle\rangle,\\ &\langle\langle a, 1\rangle, \langle \mathsf{name} = \text{“Pat”}, \mathsf{tasks} = \langle d, 1\rangle\rangle\rangle,\\ &\langle\langle a, 4\rangle, \langle \mathsf{name} = \text{“Sue”}, \mathsf{tasks} = \langle d, 2\rangle\rangle\rangle]\\ r_3' = [&\langle\langle b, 1\rangle, \text{“build”}\rangle, \langle\langle b, 2\rangle, \text{“call”}\rangle, \langle\langle b, 2\rangle, \text{“enthuse”}\rangle,\\ &\langle\langle b, 3\rangle, \text{“call”}\rangle, \langle\langle d, 1\rangle, \text{“buy”}\rangle, \langle\langle d, 2\rangle, \text{“buy”}\rangle] \end{aligned}$$

The rest of this paper gives the details of the shredding translation, explains how to stitch the results of shredded queries back together, and shows how to use row_number to avoid the space overhead of indexes. We will return to the above example throughout the paper, and we will use $Q_{org}$, $Q$ and other queries based on this example in the experimental evaluation.

## 4. SHREDDING TRANSLATION

As a pre-processing step, we annotate each comprehension body in a normalised term with a unique name $a$ — the static component of an index. We write the annotations as superscripts, for example:

$$\mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}^a\ M$$

In order to shred nested queries, we introduce an abstract type *Index* of *indexes* for maintaining the correspondence between outer and inner queries. An index $a \diamond d$ has a static component $a$ and a dynamic component $d$. The static component $a$ links the index to the corresponding $\mathsf{return}^a$ in the query. The dynamic component identifies the current bindings of the variables in the comprehension.

Next, we modify types so that bag types have an explicit index component and we use indexes to replace nested occurrences of bags within other bags:

$$\begin{array}{lrl} \text{Shredded types} & A, B ::= & \mathrm{Bag}\,\langle \mathit{Index}, F\rangle \\ \text{Flat types} & F ::= & O \mid \langle \overrightarrow{\ell : F}\rangle \mid \mathit{Index} \end{array}$$

We also adapt the syntax of terms to incorporate indexes. After shredding, terms will have the following forms:

$$\begin{array}{lrl} \text{Query terms} & L, M ::= & \biguplus \vec{C} \\ \text{Comprehensions} & C ::= & \mathsf{return}^a\ \langle I, N\rangle \\ & \mid & \mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ C \\ \text{Generators} & G ::= & x \leftarrow t \\ \text{Inner terms} & N ::= & X \mid R \mid I \\ \text{Record terms} & R ::= & \langle \overrightarrow{\ell = N}\rangle \\ \text{Base terms} & X ::= & x.\ell \mid c(\vec{X}) \mid \mathsf{empty}\ L \\ \text{Indexes} & I, J ::= & a \diamond d \\ \text{Dynamic indexes} & d ::= & \mathsf{out} \mid \mathsf{in} \end{array}$$

A comprehension is now constructed from a sequence of generator clauses of the form $\mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)$ followed by a body of the form $\mathsf{return}^a\ \langle I, N\rangle$. Each level of nesting gives rise to such a generator clause. The body always returns a pair $\langle I, N\rangle$ of an outer index $I$, denoting where the result values from the shredded query should be spliced into the final nested result, and a (flat) inner term $N$. Records are restricted to contain inner terms, which are either base types, records, or indexes, which replace nested multisets. We assume a distinguished top level static index $\top$, which allows us to treat all levels uniformly. Each shredded term is associated with an outer index out and an inner index in. (In fact out only appears in the left component of a comprehension body, and in only appears in the right component of a comprehension body. These properties will become apparent when we specify the shredding transformation on terms.)

### 4.1 Shredding types and terms

We use *paths* to point to parts of types.

$$\text{Paths} \quad p ::= \epsilon \mid \downarrow.p \mid \ell.p$$

The empty path is written $\epsilon$. A path $p$ can be extended by traversing a bag constructor ($\downarrow.p$) or selecting a label ($\ell.p$). We will sometimes write $p.\!\downarrow$ for the path $p$ with $\downarrow$ appended at the end and similarly for $p.\ell$; likewise, we will write $p.\vec{\ell}$ for the path $p$ with all the labels of $\vec{\ell}$ appended. The function $\mathit{paths}(A)$ defines the set of paths to bag types in a type $A$:

$$\begin{aligned} \mathit{paths}(O) &= \{\} \\ \mathit{paths}(\langle \ell_i : A_i\rangle_{i=1}^n) &= \textstyle\bigcup_{i=1}^n \{\ell_i.p \mid p \leftarrow \mathit{paths}(A_i)\} \\ \mathit{paths}(\mathrm{Bag}\ A) &= \{\epsilon\} \cup \{\downarrow.p \mid p \leftarrow \mathit{paths}(A)\} \end{aligned}$$

We now define a shredding translation on types. This is defined in terms of the *inner shredding* $\llbracket A \rrbracket$, a flat type that represents the contents of a bag.

$$\begin{aligned} \llbracket O \rrbracket &= O \\ \llbracket \langle \ell_i : A_i\rangle_{i=1}^n \rrbracket &= \langle \ell_i : \llbracket A_i \rrbracket\rangle_{i=1}^n \\ \llbracket \mathrm{Bag}\ A \rrbracket &= \mathit{Index} \end{aligned}$$

$$
\begin{aligned}
\lceil\!\lceil L \rceil\!\rceil_p &= \biguplus (\lceil\!\lceil L \rceil\!\rceil^{\star}_{\top,p}) \\
\lceil\!\lceil \biguplus_{i=1}^{n} C_i \rceil\!\rceil^{\star}_{a,p} &= \mathit{concat}([\lceil\!\lceil C_i \rceil\!\rceil^{\star}_{a,p}]_{i=1}^{n}) \\
\lceil\!\lceil \langle \ell_i = M_i \rangle_{i=1}^{n} \rceil\!\rceil^{\star}_{a,\ell_j.p} &= \lceil\!\lceil M_j \rceil\!\rceil^{\star}_{a,p} \\
\lceil\!\lceil \mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}^b\ M \rceil\!\rceil^{\star}_{a,\epsilon} &= [\mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}^b\ \langle a \diamond \mathsf{out}, \lfloor\!\lfloor M \rfloor\!\rfloor_b \rangle] \\
\lceil\!\lceil \mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}^b\ M \rceil\!\rceil^{\star}_{a,\downarrow.p} &= [\mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ C \mid C \leftarrow \lceil\!\lceil M \rceil\!\rceil^{\star}_{b,p}]
\end{aligned}
$$

$$
\begin{aligned}
\lfloor\!\lfloor x.\ell \rfloor\!\rfloor_a &= x.\ell \\
\lfloor\!\lfloor c([X_i]_{i=1}^{n}) \rfloor\!\rfloor_a &= c([\lfloor\!\lfloor X_i \rfloor\!\rfloor_a]_{i=1}^{n}) \\
\lfloor\!\lfloor \mathsf{empty}\ L \rfloor\!\rfloor_a &= \mathsf{empty}\ \lceil\!\lceil L \rceil\!\rceil_{\epsilon} \\
\lfloor\!\lfloor \langle \ell_i = M_i \rangle_{i=1}^{n} \rfloor\!\rfloor_a &= \langle \ell_i = \lfloor\!\lfloor M_i \rfloor\!\rfloor_a \rangle_{i=1}^{n} \\
\lfloor\!\lfloor L \rfloor\!\rfloor_a &= a \diamond \mathsf{in}
\end{aligned}
$$

**Figure 4: Shredding translation on terms**

Given a path $p \in \mathit{paths}(A)$, the type $\lceil\!\lceil A \rceil\!\rceil_p$ is the *outer shredding* of $A$ at $p$. It corresponds to the bag at path $p$ in $A$.

$$
\begin{aligned}
\lceil\!\lceil \mathrm{Bag}\ A \rceil\!\rceil_{\epsilon} &= \mathrm{Bag}\ \langle \mathit{Index}, \lfloor\!\lfloor A \rfloor\!\rfloor \rangle \\
\lceil\!\lceil \mathrm{Bag}\ A \rceil\!\rceil_{\downarrow.p} &= \lceil\!\lceil A \rceil\!\rceil_p \\
\lceil\!\lceil \langle \overrightarrow{\ell : A} \rangle \rceil\!\rceil_{\ell_i.p} &= \lceil\!\lceil A_i \rceil\!\rceil_p
\end{aligned}
$$

For example, consider the result type *Result* from Section 3. Its nesting degree is 3, and its paths are:

$$\mathit{paths}(\mathit{Result}) = \{\epsilon, \downarrow.\mathsf{people}.\epsilon, \downarrow.\mathsf{people}.\downarrow.\mathsf{tasks}.\epsilon\}$$

We can shred *Result* in three ways using these three paths, yielding three shredded types:

$$
\begin{aligned}
A_1 &= \lceil\!\lceil \mathit{Result} \rceil\!\rceil_{\epsilon} \\
A_2 &= \lceil\!\lceil \mathit{Result} \rceil\!\rceil_{\downarrow.\mathsf{people}.\epsilon} \\
A_3 &= \lceil\!\lceil \mathit{Result} \rceil\!\rceil_{\downarrow.\mathsf{people}.\epsilon.\downarrow.\mathsf{tasks}.\epsilon}
\end{aligned}
$$

or equivalently:

$$
\begin{aligned}
A_1 &= \mathrm{Bag}\ \langle \mathit{Index}, \langle \mathsf{department} : \mathit{String}, \mathsf{people} : \mathit{Index} \rangle \rangle \\
A_2 &= \mathrm{Bag}\ \langle \mathit{Index}, \langle \mathsf{name} : \mathit{String}, \mathsf{tasks} : \mathit{Index} \rangle \rangle \\
A_3 &= \mathrm{Bag}\ \langle \mathit{Index}, \mathit{String} \rangle
\end{aligned}
$$

The shredding translation on terms $\lceil\!\lceil L \rceil\!\rceil_p$ is given in Figure 4. This takes a term $L$ and a path $p$ and gives a query $\lceil\!\lceil L \rceil\!\rceil_p$ that computes a result of type $\lceil\!\lceil A \rceil\!\rceil_p$, where $A$ is the type of $L$. The auxiliary translation $\lceil\!\lceil M \rceil\!\rceil^{\star}_{a,p}$ returns the shredded comprehensions of $M$ along path $p$ with outer static index $a$. The auxiliary translation $\lfloor\!\lfloor M \rfloor\!\rfloor_a$ produces a flat representation of $M$ with inner static index $a$. Note that the shredding translation is linear in time and space. Observe that for emptiness tests we need only the top-level query.

Continuing the example, we can shred $Q_{comp}$ in three ways, yielding shredded queries:

$$
\begin{aligned}
q_1 &= \lceil\!\lceil Q_{comp} \rceil\!\rceil_{\epsilon} \\
q_2 &= \lceil\!\lceil Q_{comp} \rceil\!\rceil_{\downarrow.\mathsf{people}.\epsilon} \\
q_3 &= \lceil\!\lceil Q_{comp} \rceil\!\rceil_{\downarrow.\mathsf{people}.\epsilon.\downarrow.\mathsf{tasks}.\epsilon}
\end{aligned}
$$

or equivalently:

$$
\begin{aligned}
q_1 = {} & \mathsf{for}\ (x \leftarrow \mathit{departments}) \\
& \quad \mathsf{return}^a\ \langle \top \diamond 1, \langle \mathsf{department} = x.\mathsf{name}, \mathsf{people} = a \diamond \mathsf{in} \rangle \rangle \\
q_2 = {} & (\mathsf{for}\ (x \leftarrow \mathit{departments}) \\
& \quad \mathsf{for}\ (y \leftarrow \mathit{employees})\ \mathsf{where}\ (x.\mathsf{name} = y.\mathsf{dept} \land \\
& \qquad\qquad (y.\mathsf{salary} < 1000 \lor y.\mathsf{salary} > 1000000)) \\
& \quad \mathsf{return}^b\ (\langle a \diamond \mathsf{out}, \langle \mathsf{name} = y.\mathsf{name}, \mathsf{tasks} = b \diamond \mathsf{in} \rangle \rangle)) \\
& \uplus\ (\mathsf{for}\ (x \leftarrow \mathit{departments}) \\
& \quad \mathsf{for}\ (y \leftarrow \mathit{contacts})\ \mathsf{where}\ (x.\mathsf{name} = y.\mathsf{dept} \land y.\mathsf{client}) \\
& \quad \mathsf{return}^d\ (\langle a \diamond \mathsf{out}, \langle \mathsf{name} = y.\mathsf{name}, \mathsf{tasks} = d \diamond \mathsf{in} \rangle \rangle)) \\
q_3 = {} & (\mathsf{for}\ (x \leftarrow \mathit{departments}) \\
& \quad \mathsf{for}\ (y \leftarrow \mathit{employees})\ \mathsf{where}\ (x.\mathsf{name} = y.\mathsf{dept} \land \\
& \qquad\qquad (y.\mathsf{salary} < 1000 \lor y.\mathsf{salary} > 1000000)) \\
& \quad \mathsf{for}\ (z \leftarrow \mathit{tasks})\ \mathsf{where}\ (z.\mathsf{employee} = y.\mathsf{employee}) \\
& \quad \mathsf{return}^c\ \langle b \diamond \mathsf{out}, z.\mathsf{task} \rangle) \\
& \uplus\ (\mathsf{for}\ (x \leftarrow \mathit{departments}) \\
& \quad \mathsf{for}\ (y \leftarrow \mathit{contacts})\ \mathsf{where}\ (x.\mathsf{name} = y.\mathsf{dept} \land y.\mathsf{client}) \\
& \quad \mathsf{return}^e\ \langle d \diamond \mathsf{out}, \text{“buy”} \rangle)
\end{aligned}
$$

As a sanity check, we show that well-formed normalised terms shred to well-formed shredded terms of the appropriate shredded types. We discuss other correctness properties of the shredding translation in Section 5.

THEOREM 2. *Suppose $L$ is a normalised flat-nested query with $\vdash L : A$ and $p \in \mathit{paths}(A)$, then $\vdash \lceil\!\lceil L \rceil\!\rceil_p : \lceil\!\lceil A \rceil\!\rceil_p$.*

## 4.2 Shredded packages

To maintain the relationship between shredded terms and the structure of the nested result they are meant to construct, we use *shredded packages*. A shredded package $\hat{A}$ is a nested type with annotations, denoted $(-)^{\alpha}$, attached to each bag constructor.

$$\hat{A} ::= O \mid \langle \overrightarrow{\ell : \hat{A}} \rangle \mid (\mathrm{Bag}\ \hat{A})^{\alpha}$$

For a given package, the annotations are drawn from the same set. We write $\hat{A}(S)$ to denote a shredded package with annotations drawn from the set $S$. We sometimes omit the type parameter when it is clear from context.

Given a shredded package $\hat{A}$, we can erase its annotations to obtain its underlying type.

$$
\begin{aligned}
\mathit{erase}(O) &= O \\
\mathit{erase}(\langle \ell_i : \hat{A}_i \rangle_{i=1}^{n}) &= \langle \ell_i : \mathit{erase}(\hat{A}_i) \rangle_{i=1}^{n} \\
\mathit{erase}((\mathrm{Bag}\ \hat{A})^{\alpha}) &= \mathrm{Bag}\ (\mathit{erase}(\hat{A}))
\end{aligned}
$$

We lift the type shredding function $\lceil\!\lceil A \rceil\!\rceil$ to produce a shredded package $\mathit{Shred}(A)$, where each annotation contains the shredded version of the input type or query along the path to the associated bag constructor. We define $\mathit{Shred}(A) = \mathit{Shred}_{\epsilon}(A)$ as follows:

$$
\begin{aligned}
\mathit{Shred}_p(O) &= O \\
\mathit{Shred}_p(\langle \ell_i : A_i \rangle_{i=1}^{n}) &= \langle \ell_i : \mathit{Shred}_{p.\ell_i}(A_i) \rangle_{i=1}^{n} \\
\mathit{Shred}_p(\mathrm{Bag}\ A) &= (\mathrm{Bag}\ (\mathit{Shred}_{p.\downarrow}(A)))^{\lceil\!\lceil B \rceil\!\rceil_p}
\end{aligned}
$$

Similarly, we lift the term shredding function$\lceil\!\lceil L \rceil\!\rceil$ to produce a shredded query package $\mathit{shred}(L) = \mathit{shred}_{\epsilon}(L)$, where:

$$
\begin{aligned}
\mathit{shred}_p(O) &= O \\
\mathit{shred}_p(\langle \ell_i : A_i \rangle_{i=1}^{n}) &= \langle \ell_i : \mathit{shred}_{p.\ell_i}(A_i) \rangle_{i=1}^{n} \\
\mathit{shred}_p(\mathrm{Bag}\ A) &= (\mathrm{Bag}\ (\mathit{shred}_{p.\downarrow}(A)))^{\lceil\!\lceil L \rceil\!\rceil_p}
\end{aligned}
$$

For example, the shredded package for the *Result* type from Section 3 is:

$$
\begin{aligned}
& \mathit{Shred}_{\mathit{Result}}(\mathit{Result}) = \\
& \quad \mathrm{Bag}\ \langle \mathsf{department} : \mathit{String}, \\
& \qquad\quad \mathsf{people} : \mathrm{Bag}\ \langle \mathsf{name} : \mathit{String}, \\
& \qquad\qquad\qquad \mathsf{tasks} : (\mathrm{Bag}\ \mathit{String})^{A_3} \rangle^{A_2} \rangle^{A_1}
\end{aligned}
$$

where $A_1, A_2$, and $A_3$ are as shown in Section 4.1. Shredding the normalised query $Q'$ gives the same package, except the type annotations $A_1, A_2, A_3$ become queries $q_1, q_2, q_3$.

Again, as a sanity check we show that erasure is the left inverse of type shredding and that term-level shredding operations preserve types.

THEOREM 3. *For any type $A$, we have $\mathit{erase}(\mathit{shred}_A(A)) = A$. Furthermore, if $L$ is a closed, normalised, flat–nested query such that $\vdash L : A$ then $\vdash \mathit{shred}_L(A) : \mathit{shred}_A(A)$.*

## 5. QUERY EVALUATION AND STITCHING

Having shredded a normalised nested query, we can then run all of the resulting shredded queries separately. If we stitch the shredded results together to form a nested result, then we obtain the same nested result as we would obtain by running the nested query directly. In this section we describe how to run shredded queries and stitch their results back together to form a nested result.

### 5.1 Evaluating shredded queries

The semantics of shredded queries $\mathcal{S}[\![-]\!]$ is given in Figure 5. Apart from the handling of indexes, it is much like the semantics for nested queries given in Figure 2. To allow different implementations of indexes, we first define a canonical notion of index, and then parameterise the semantics by the concrete type of indexes $X$ and a function $\mathit{index} : \mathit{Index} \to X$ mapping each canonical index to a distinct concrete index. A canonical index $a \diamond \iota$ comprises static index $a$ and dynamic index $\iota$, where the latter is a list of positive natural numbers. For now we take concrete indexes simply to be canonical indexes, and $\mathit{index}$ to be the identity function. We consider other definitions of $\mathit{index}$ in Section 6.

The current dynamic index $\iota$ is threaded through the semantics in tandem with the environment $\rho$. The former encodes the position of each of the generators in the current comprehension and allows us to invoke $\mathit{index}$ to construct a concrete index. The outer index at dynamic index $\iota.i$ is $\iota$; the inner index is $\iota.i$. In order for a comprehension to generate dynamic indexes we use the function $\mathit{enum}$ (introduced in Section 2) that takes a list of elements and returns the same list with the element number paired up with each source element.

Running a shredded query yields a list of pairs of indexes and shredded values.

| | |
|---|---|
| Results | $s ::= [\langle I_1, w_1\rangle, \ldots, \langle I_m, w_m\rangle]$ |
| Flat values | $w ::= c \mid \langle \ell_1 = w_1, \ldots, \ell_n = w_n\rangle \mid I$ |

Given a shredded package $\hat{A}(S)$ and a function $f : S \to T$, we can map $f$ over the annotations to obtain a new shredded package $\hat{A}'(T)$ such that $\mathit{erase}(\hat{A}) = \mathit{erase}(\hat{A}')$.

$$\begin{aligned} \mathit{pmap}_f(O) &= O \\ \mathit{pmap}_f(\langle \ell_i : \hat{A}_i\rangle_{i=1}^n) &= \langle \ell_i : \mathit{pmap}_f(\hat{A}_i)\rangle_{i=1}^n \\ \mathit{pmap}_f((\mathrm{Bag}\,\hat{A})^\alpha) &= ((\mathrm{Bag}\,\mathit{pmap}_f(\hat{A})))^{f(\alpha)} \end{aligned}$$

The semantics of a shredded query package is a *shredded value package* containing indexed results for each shredded query. For each type $A$ we define $\mathcal{H}[\![A]\!] = \mathit{shred}_A(A)$ and for each flat–nested, closed $\vdash L : A$ we define $\mathcal{H}[\![L]\!]_A : \mathcal{H}[\![A]\!]$ as $\mathit{pmap}_{\mathcal{S}[\![-]\!]}(\mathit{shred}_L(A))$. In other words, we first construct the shredded query package $\mathit{shred}_L(A)$, then apply the shredded semantics $\mathcal{S}[\![q]\!]$ to each query $q$ in the package.

For example, here is the shredded package that we obtain after running the normalised query $Q_{comp}$ from Section 2.2:

$$\begin{aligned} \mathcal{H}[\![Q_{comp}]\!]_A = \mathrm{Bag}\,\langle &\mathsf{department} : \mathit{String}, \\ &\mathsf{people} : \mathrm{Bag}\,\langle \mathsf{name} : \mathit{String}, \\ &\qquad \mathsf{tasks} : (\mathrm{Bag}\,\mathit{String})^{r_3}\rangle^{r_2}\rangle^{r_1} \end{aligned}$$

where $r_1$, $r_2$, and $r_3$ are as in Section 3 except that indexes are of the form $a \diamond 1.2.3$ instead of $\langle a, 1, 2, 3\rangle$.

### 5.2 Stitching shredded query results together

A shredded value package can be *stitched* back together into a nested value, preserving annotations, as follows:

$$\begin{aligned} \mathit{stitch}(\hat{A}) &= \mathit{stitch}_{\top \diamond 1}(\hat{A}) \\ \mathit{stitch}_c(O) &= c \\ \mathit{stitch}_r(\langle \ell_i : \hat{A}_i\rangle_{i=1}^n) &= \langle \ell_i = \mathit{stitch}_{r.\ell_i}(\hat{A}_i)\rangle_{i=1}^n \\ \mathit{stitch}_I((\mathrm{Bag}\,\hat{A})^s) &= [(\mathit{stitch}_w(\hat{A})) \mid \langle I, w\rangle \leftarrow s] \end{aligned}$$

The flat value parameter $w$ to the auxiliary function $\mathit{stitch}_w(-)$ specifies which values to stitch along the current path.

Resuming our running example, once the results $r_1 : A_1, r_2 : A_2, r_3 : A_3$ have been evaluated on the database, they are shipped back to the host system where we can run the following code in-memory to stitch the three tables together into a single value: the result of the original nested query. The code for this query $Q_{stitch}$ follows the same idea as the query $Q_{org}$ that constructs the nested $\mathit{organisation}$ from $\Sigma$.

```
for ($x \leftarrow r_1$)
return ($\langle$department $= x$.name,
        people $=$ for ($\langle i, y\rangle \leftarrow r_2$)
                   where ($x$.people $= i$))
                   return ($\langle$name $= y$.name,
                           tasks $=$ for ($\langle j, z\rangle \leftarrow r_3$)
                                     where ($y$.tasks $= j$)
                                     return $z$$\rangle$)$\rangle$))
```

We can now state our key correctness property: evaluating shredded queries and stitching the results back together yields the same results as evaluating the original nested query directly.

THEOREM 4. *If* $\vdash L : \mathrm{Bag}\,A$ *then:*

$$\mathit{stitch}(\mathcal{H}[\![L]\!]_{\mathrm{Bag}\,A}) = \mathcal{N}[\![L]\!]$$

PROOF SKETCH. We omit the full proof due to space limits; it is available in the full version of this paper. The proof introduces several intermediate definitions. Specifically, we consider an *annotated semantics* for nested queries in which each collection element is tagged with an index, and we show that this semantics is consistent with the ordinary semantics if the annotations are erased. We then prove the correctness of shredding and stitching with respect to the annotated semantics, and the desired result follows. □

## 6. INDEXING SCHEMES

So far, we have worked with canonical indexes of the form $a \diamond 1.2.3$. These could be represented in SQL by using multiple columns (padding with NULLs if necessary) since for a given query the length of the dynamic part is bounded by the number of for-comprehensions in the query. This imposes space and running time overhead due to constructing and maintaining the indexes. Instead, in this section we consider alternative, more compact indexing schemes.

We can define alternative indexing schemes by varying the $\mathit{index}$ parameter of the shredded semantics (see Section 5.1). Not all possible instantiations are valid. To identify those that are, we first define a function for computing the canonical indexes of a nested query result.

$$\begin{aligned} \mathcal{I}[\![L]\!] &= \mathcal{I}[\![L]\!]_{\varepsilon,1} \\ \mathcal{I}[\![\textstyle\biguplus_{i=1}^n C_i]\!]_{\rho,\iota} &= \mathit{concat}([\mathcal{I}[\![C_i]\!]_{\rho,\iota}]_{i=1}^n) \\ \mathcal{I}[\![\langle \ell_i = M_i\rangle_{i=1}^n]\!]_{\rho,\iota} &= \mathit{concat}([\mathcal{I}[\![M_i]\!]_{\rho,\iota}]_{i=1}^n) \\ \mathcal{I}[\![X]\!]_{\rho,\iota} &= [\,] \end{aligned}$$

$$\begin{aligned} &\mathcal{I}[\![\mathsf{for}\ ([x_i \leftarrow t_i]_{i=1}^n\ \mathsf{where}\ X)\ \mathsf{return}^a\ M]\!]_{\rho,\iota} = \\ &\quad \mathit{concat}([a \diamond \iota.j :: \mathcal{I}[\![M]\!]_{\rho[x_i \mapsto r_i]_{i=1}^n, \iota.j} \\ &\qquad \mid \langle j, \vec{r}\rangle \leftarrow \mathit{enum}([\vec{r} \mid [r_i \leftarrow [\![t_i]\!]]_{i=1}^n, \mathcal{N}[\![X]\!]_{\rho[x_i \mapsto r_i]_{i=1}^n}])]) \end{aligned}$$

An indexing function $\mathit{index} : \mathit{Index} \to X$ is *valid* with respect to the closed nested query $L$ if it is injective and defined on every canonical index in $\mathcal{I}[\![L]\!]$. The only requirement on indexes in the proof of Theorem 4 is that $\mathit{index}$ is valid. We consider two alternative valid indexing schemes: *natural* and *flat indexes*.

$$\mathcal{S}\llbracket L \rrbracket = \mathcal{S}\llbracket L \rrbracket_{\varepsilon,1} \qquad \begin{array}{c}\mathcal{S}\llbracket \langle \ell = N\rangle_{i=1}^{n} \rrbracket_{\rho,\iota} = \langle \ell_i = \mathcal{S}\llbracket N_i \rrbracket_{\rho,\iota}\rangle_{i=1}^{n} \\ \mathcal{S}\llbracket X \rrbracket_{\rho,\iota} = \mathcal{N}\llbracket X \rrbracket_{\rho}\end{array} \qquad \begin{array}{r}\mathcal{S}\llbracket a \diamond \mathsf{out} \rrbracket_{\rho,\iota.i} = \mathit{index}(a \diamond \iota) \\ \mathcal{S}\llbracket a \diamond \mathsf{in} \rrbracket_{\rho,\iota.i} = \mathit{index}(a \diamond \iota.i)\end{array}$$

$$\mathcal{S}\llbracket \biguplus_{i=1}^{n} C_i \rrbracket_{\rho,\iota} = \mathit{concat}([\mathcal{S}\llbracket C_i \rrbracket_{\rho,\iota}]_{i=1}^{n}) \qquad \mathcal{S}\llbracket \mathsf{return}^{a}\, N \rrbracket_{\rho,\iota} = [\mathcal{S}\llbracket N \rrbracket_{\rho,\iota}]$$

$$\mathcal{S}\llbracket \mathsf{for}\ ([x_i \leftarrow t_i]_{i=1}^{n}\ \mathsf{where}\ X)\ C \rrbracket_{\rho,\iota} = \mathit{concat}([\mathcal{S}\llbracket C \rrbracket_{\rho[x_i \mapsto r_i]_{i=1}^{n},\iota.j} \mid \langle j, \vec{r}\rangle \leftarrow \mathit{enum}([\vec{r} \mid [r_i \leftarrow \llbracket t_i \rrbracket]_{i=1}^{n}, \mathcal{N}\llbracket X \rrbracket_{\rho[x_i \mapsto r_i]_{i=1}^{n}}])])$$

**Figure 5: Semantics of shredded queries**

## 6.1 Natural indexes

Natural indexes are synthesised from row data. In order to generate a natural index for a query every table must have a key, that is, a collection of fields guaranteed to be unique for every row in the table. For sets, this is always possible by using all of the field values as a key; this idea is used in Van den Bussche's simulation for sets [26]. However, for bags this is not always possible, so using natural indexes may require adding extra key fields.

Given a table $t$, let $key_t$ be the function that given a row $r$ of $t$ returns the key fields of $r$. We now define a function to compute the list of natural indexes for a query $L$.

$$\begin{array}{rl}
\mathcal{I}^{\natural}\llbracket L \rrbracket &= \mathcal{I}^{\natural}\llbracket L \rrbracket_{\varepsilon} \\
\mathcal{I}^{\natural}\llbracket \biguplus_{i=1}^{n} C_i \rrbracket_{\rho} &= \mathit{concat}([\mathcal{I}^{\natural}\llbracket C_i \rrbracket_{\rho}]_{i=1}^{n}) \\
\mathcal{I}^{\natural}\llbracket \langle \ell_i = M_i\rangle_{i=1}^{n} \rrbracket_{\rho} &= \mathit{concat}([\mathcal{I}^{\natural}\llbracket M_i \rrbracket_{\rho}]_{i=1}^{n}) \\
\mathcal{I}^{\natural}\llbracket X \rrbracket_{\rho} &= [\,]
\end{array}$$

$$\begin{array}{l}
\mathcal{I}^{\natural}\llbracket \mathsf{for}\ ([x_i \leftarrow t_i]_{i=1}^{n}\ \mathsf{where}\ X)\ \mathsf{return}^{a}\ M \rrbracket_{\rho} = \\
\quad \mathit{concat}([a \diamond \langle key_{t_i}(r_i)\rangle_{i=1}^{n} :: \mathcal{I}^{\natural}\llbracket M \rrbracket_{\rho[x_i \mapsto r_i]_{i=1}^{n}} \\
\qquad\qquad \mid [r_i \leftarrow \llbracket t_i \rrbracket]_{i=1}^{n}, \mathcal{N}\llbracket X \rrbracket_{\rho[x_i \mapsto r_i]_{i=1}^{n}}])
\end{array}$$

If $a \diamond \iota$ is the $i$-th element of $\mathcal{I}\llbracket L \rrbracket$, then $index_L^{\natural}(a \diamond \iota)$ is defined as the $i$-th element of $\mathcal{I}^{\natural}\llbracket L \rrbracket$. The natural indexing scheme is defined by setting $index = index_L^{\natural}$.

An advantage of natural indexes is that they can be implemented in plain SQL, so for a given comprehension all where clauses can be amalgamated (using the $\wedge$ operator) and no auxiliary subqueries are needed. The downside is that the type of a dynamic index may still vary across the component comprehensions of a shredded query, complicating implementation of the query (due to the need to pad some subqueries with null columns) and potentially decreasing performance due to increased data movement. We now consider an alternative, in which row_number is used to generate dynamic indexes.

## 6.2 Flat indexes and let-insertion

The idea of flat indexes is to enumerate all of the canonical dynamic indexes associated with each static index and use the enumeration as the dynamic index.

Let $\iota$ be the $i$-th element of the list $[\iota' \mid a \diamond \iota' \leftarrow \mathcal{I}\llbracket L \rrbracket]$, then $index_L^{\flat}(a \diamond \iota) = \langle a, i\rangle$. The flat indexing scheme is defined by setting $index = index_L^{\flat}$. Let $\mathcal{I}^{\flat}\llbracket L \rrbracket = [index_L^{\flat}(I) \mid I \leftarrow \mathcal{I}\llbracket L \rrbracket]$ and let $\mathcal{S}^{\flat}\llbracket L \rrbracket$ be $\mathcal{S}\llbracket L \rrbracket$ where $index = index_L^{\flat}$.

In this section, we give a translation called let-insertion that uses let-binding and an index primitive to manage flat indexes. In the next section, we take the final step from this language to SQL.

Our semantics for shredded queries uses canonical indexes. We now specify a target language providing flat indexes. In order to do so, we introduce let-bound sub-queries, and translate each comprehension into the following form:

$$\begin{array}{l}
\mathsf{let}\ q = \mathsf{for}\ (\overrightarrow{G_{\mathsf{out}}}\ \mathsf{where}\ X_{\mathsf{out}})\ \mathsf{return}\ N_{\mathsf{out}}\ \mathsf{in} \\
\quad \mathsf{for}\ (\overrightarrow{G_{\mathsf{in}}}\ \mathsf{where}\ X_{\mathsf{in}})\ \mathsf{return}\ N_{\mathsf{in}}
\end{array}$$

The special index expression is available in each loop body, and is bound to the current index value.

Following let-insertion, the types are as before, except indexes are represented as pairs of integers $\langle a, d\rangle$ where $a$ is the static component and $d$ is the dynamic component.

$$\begin{array}{lrl}
\text{Types} & A, B & ::= \mathrm{Bag}\,\langle\langle \mathit{Int}, \mathit{Int}\rangle, F\rangle \\
\text{Flat types} & F & ::= O \mid \langle \overrightarrow{\ell : F}\rangle \mid \langle \mathit{Int}, \mathit{Int}\rangle
\end{array}$$

The syntax of terms is adapted as follows:

$$\begin{array}{lrl}
\text{Query terms} & L, M & ::= \biguplus \vec{C} \\
\text{Comprehensions} & C & ::= \mathsf{let}\ q = S\ \mathsf{in}\ S' \\
\text{Subqueries} & S & ::= \mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}\ N \\
\text{Data sources} & u & ::= t \mid q \\
\text{Generators} & G & ::= x \leftarrow u \\
\text{Inner terms} & N & ::= X \mid R \mid \mathsf{index} \\
\text{Record terms} & R & ::= \langle \overrightarrow{\ell = N}\rangle \\
\text{Base terms} & X & ::= x.\ell \mid c(\vec{X}) \mid \mathsf{empty}\ L
\end{array}$$

The semantics of let-inserted queries is given in Figure 6. Rather than maintaining a canonical index, it generates a flat index for each subquery.

We first give the translation on shredded types as follows:

$$\begin{array}{rl}
\mathbf{L}(O) &= O \\
\mathbf{L}(\langle \overrightarrow{\ell : F}\rangle) &= \langle \overrightarrow{\ell : \mathbf{L}(F)}\rangle \\
\mathbf{L}(\mathit{Index}) &= \langle \mathit{Int}, \mathit{Int}\rangle \\
\mathbf{L}(\mathrm{Bag}\,\langle \mathit{Index}, F\rangle) &= \mathrm{Bag}\,\langle\langle \mathit{Int}, \mathit{Int}\rangle, \mathbf{L}(F)\rangle
\end{array}$$

For example:

$$\mathbf{L}(A_2) = \mathrm{Bag}\,\langle\langle \mathit{Int}, \mathit{Int}\rangle, \langle \mathrm{name} : \mathit{String}, \mathrm{tasks} : \langle \mathit{Int}, \mathit{Int}\rangle\rangle\rangle$$

Without loss of generality we rename all the bound variables in our source query to ensure that all bound variables have distinct names, and that none coincides with the distinguished name $z$ used for let-bindings. The let-insertion translation $\mathbf{L}$ is defined in Figure 7, where we use the following auxiliary functions:

$$\begin{array}{rl}
\mathit{expand}(x, t) &= \langle \ell_i = x.\ell_i\rangle_{i=1}^{n} \\
 & \qquad \text{where } \Sigma(t) = \mathrm{Bag}\,\langle \overrightarrow{\ell : A}\rangle \\
\mathit{gens}\ (\mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ C) &= \vec{G} :: \mathit{gens}\ C \\
\mathit{gens}\ (\mathsf{return}^{a}\ N) &= [\,] \\
\mathit{conds}\ (\mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ C) &= X :: \mathit{conds}\ C \\
\mathit{conds}\ (\mathsf{return}^{a}\ N) &= [\,] \\
\mathit{body}\ (\mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ C) &= \mathit{body}\ C \\
\mathit{body}\ (\mathsf{return}^{a}\ N) &= N
\end{array}$$

Each comprehension is rearranged into two sub-queries. The first generates the outer indexes. The second computes the results. The translation sometimes produces $n$-ary projections in order to refer to values bound by the first subquery inside the second.

For example, applying $\mathbf{L}$ to $q_1$ from Section 4.2 yields:

$$\begin{array}{l}
\mathsf{for}\ (x \leftarrow \mathit{departments}) \\
\mathsf{return}\ \langle\langle \top, 1\rangle, \langle \mathrm{dept} = x.\mathrm{name}, \mathrm{people} = \mathsf{index}\rangle\rangle
\end{array}$$

$$\mathcal{L}[\![L]\!] = \mathcal{L}[\![L]\!]_\varepsilon \qquad \mathcal{L}[\![t]\!]_\rho = [\![t]\!] \qquad \mathcal{L}[\![\langle \ell_j = N_j\rangle_{j=1}^m]\!]_{\rho,i} = \langle \ell_j = \mathcal{L}[\![N_j]\!]_{\rho,i}\rangle_{j=1}^m \qquad \mathcal{L}[\![\mathsf{index}]\!]_{\rho,i} = i$$

$$\mathcal{L}[\![\biguplus_{j=1}^m C_j]\!]_\rho = concat([\mathcal{L}[\![C_j]\!]_\rho]_{j=1}^m) \qquad \mathcal{L}[\![q]\!]_\rho = \rho(q) \qquad \mathcal{L}[\![X]\!]_{\rho,i} = \mathcal{N}[\![X]\!]_\rho$$

$$\mathcal{L}[\![\mathsf{let}\; q = S_{\mathsf{out}}\;\mathsf{in}\; S_{\mathsf{in}}]\!]_\rho = \mathcal{L}[\![S_{\mathsf{in}}]\!]_{\rho[q \mapsto \mathcal{L}[\![S_{\mathsf{out}}]\!]_\rho]}$$

$$\mathcal{L}[\![\mathsf{for}\;([x_j \leftarrow u_j]_{j=1}^m\;\mathsf{where}\;X)\;\mathsf{return}\;N]\!]_\rho = [\mathcal{L}[\![N]\!]_{\rho[x_j \mapsto r_j]_{j=1}^m, i} \mid \langle i, \vec{r}\rangle \leftarrow enum([\vec{r} \mid [r_j \leftarrow \mathcal{L}[\![u_j]\!]_\rho]_{j=1}^m, \mathcal{N}[\![X]\!]_{\rho[x_j \mapsto r_j]_{j=1}^m}])]$$

**Figure 6: Semantics of let-inserted shredded queries**

$$\mathbf{L}(\biguplus_{i=1}^n C_i) = \biguplus_{i=1}^n \mathbf{L}(C_i)$$

$$\mathbf{L}(C) = \mathsf{let}\; q = (\mathsf{for}\;(\overrightarrow{G_{\mathsf{out}}}\;\mathsf{where}\;X_{\mathsf{out}})\;\mathsf{return}\;\langle R_{\mathsf{out}}, \mathsf{index}\rangle)\;\mathsf{in}\;\mathsf{for}\;(z \leftarrow q, \overrightarrow{G_{\mathsf{in}}}\;\mathsf{where}\;\mathbf{L}_{\vec{y}}(X_{\mathsf{in}}))\;\mathsf{return}\;\mathbf{L}_{\vec{y}}(N)$$

$$\text{where}\quad \overrightarrow{G_{\mathsf{out}}} = concat\;(init\;(gens\;C)) \qquad \overrightarrow{y = t} = \overrightarrow{G_{\mathsf{out}}} \qquad \overrightarrow{G_{\mathsf{in}}} = last\;(gens\;C) \qquad N = body\;C$$

$$X_{\mathsf{out}} = \bigwedge init\;(conds\;C) \qquad R_{\mathsf{out}} = \langle expand(y_i, t_i)\rangle_{i=1}^n \qquad X_{\mathsf{in}} = last\;(conds\;C) \qquad n = length\;\overrightarrow{G_{\mathsf{out}}}$$

$$\mathbf{L}_{\vec{y}}(x.\ell) = \begin{cases} x.\ell, & \text{if } x \notin \{y_1, \dots, y_n\} \\ z.1.i.\ell, & \text{if } x = y_i \end{cases}$$

$$\mathbf{L}_{\vec{y}}(c(X_1, \dots, X_m)) = c(\mathbf{L}_{\vec{y}}(X_1), \dots, \mathbf{L}_{\vec{y}}(X_m))$$

$$\mathbf{L}_{\vec{y}}(\langle \ell_j = X_j\rangle_{j=1}^m) = \langle \ell_j = \mathbf{L}_{\vec{y}}(X_j)\rangle_{j=1}^m$$

$$\mathbf{L}_{\vec{y}}(a \diamond d) = \langle a, \mathbf{L}(d)\rangle$$

$$\mathbf{L}_{\vec{y}}(\mathsf{empty}\;L) = \mathsf{empty}\;(\mathbf{L}_{\vec{y}}(L))$$

$$\mathbf{L}_{\vec{y}}(\biguplus_{i=1}^n C_i) = \biguplus_{i=1}^n \mathbf{L}_{\vec{y}}(C_i)$$

$$\mathbf{L}_{\vec{y}}(\mathsf{for}\;(\vec{G}\;\mathsf{where}\;X)\;\mathsf{return}^a\;\langle a, N\rangle) = \mathsf{for}\;(\vec{G}\;\mathsf{where}\;\mathbf{L}_{\vec{y}}(X))\;\mathsf{return}\;\langle a, \mathbf{L}_{\vec{y}}(N)\rangle$$

$$\mathbf{L}(\mathsf{out}) = z.2 \qquad \mathbf{L}(\mathsf{in}) = \mathsf{index}$$

**Figure 7: The let-insertion translation**

and $q_2$ becomes:

(let $q$ = for ($x \leftarrow$ *departments*) return $\langle\langle$dept $= x$.name$\rangle$, index$\rangle$ in
  for ($z \leftarrow q, y \leftarrow$ *employees*) where ($z$.1.1.name $= y$.dept $\wedge$
    ($y$.salary $< 1000 \vee y$.salary $> 1000000$))
  return ($\langle\langle a, z.2\rangle$, $\langle$name $= y$.name, tasks $= \langle b$, index$\rangle\rangle\rangle$))
$\uplus$
(let $q$ = for ($x \leftarrow$ *departments*) return $\langle\langle$dept $= x$.name$\rangle$, index$\rangle$ in
  for ($z \leftarrow q, y \leftarrow$ *contacts*) where ($z$.1.1.name $= y$.dept $\wedge y$.client)
  return ($\langle\langle a, z.2\rangle$, $\langle$name $= y$.name, tasks $= \langle d$, index$\rangle\rangle\rangle$))

As a sanity check, we show that the translation is type-preserving:

THEOREM 5. *Given shredded query* $\vdash M : \mathrm{Bag}\,\langle Index, F\rangle$, *then* $\vdash \mathbf{L}(M) : \mathbf{L}(\mathrm{Bag}\,\langle Index, F\rangle)$.

To prove the correctness of let-insertion, we need to show that the shredded semantics and let-inserted semantics agree. In the statement of the correctness theorem, recall that $\mathcal{S}^\flat[\![L]\!]$ refers to the version of $\mathcal{S}[\![L]\!]$ using $index = index^\flat_L$.

THEOREM 6. *Suppose* $\vdash L : A$ *and* $[\![L]\!]_p = M$. *Then* $\mathcal{S}^\flat[\![M]\!] = \mathcal{L}[\![\mathbf{L}(M)]\!]$.

PROOF SKETCH. The high-level idea is to separate results into data and indexes and compare each separately. It is straightforward, albeit tedious, to show that the different definitions are equal if we replace all dynamic indexes by a dummy value. It then remains to show that the dynamic indexes agree. The pertinent case is the translation of a comprehension:

$$[\mathsf{for}\;(\vec{G_i} \leftarrow X_i)]_{i=1}^n\;\mathsf{for}\;(\vec{G_{\mathsf{in}}} \leftarrow X_{\mathsf{in}})\;\mathsf{return}^b\;\langle a \diamond \mathsf{out}, N\rangle$$

which becomes let $q = S_{\mathsf{out}}$ in $S_{\mathsf{in}}$ for suitable $S_{\mathsf{out}}$ and $S_{\mathsf{in}}$. The dynamic indexes computed by $S_{\mathsf{out}}$ coincide exactly with those of $\mathcal{I}^\flat[\![L]\!]$ at static index $a$, and the dynamic indexes computed by $S_{\mathsf{in}}$, if there are any, coincide exactly with those of $\mathcal{I}^\flat[\![L]\!]$. □

## 7. CONVERSION TO SQL

Earlier translation stages have employed nested records for convenience, but SQL does not support nested records. At this stage, we eliminate nested records from queries. For example, we can represent a nested record $\langle$a $= \langle$b $= 1$, c $= 2\rangle$, d $= 3\rangle$ as a flat record $\langle$a_b $= 1$, a_c $= 2$, d $= 3\rangle$.

In order to interpret shredded, flattened, let-inserted terms as SQL, we interpret index generators using SQL's OLAP facilities.

| | |
|---|---|
| Query terms | $L ::= (\mathsf{union\ all})\;\vec{C}$ |
| Comprehensions | $C ::= \mathsf{with}\; q\;\mathsf{as}\;(S)\;C \mid S'$ |
| Subqueries | $S ::= \mathsf{select}\; R\;\mathsf{from}\;\vec{G}\;\mathsf{where}\;X$ |
| Data sources | $u ::= t \mid q$ |
| Generators | $G ::= u\;\mathsf{as}\;x$ |
| Inner terms | $N ::= X \mid \mathsf{row\_number}()\;\mathsf{over}\;(\mathsf{order\;by}\;\vec{X})$ |
| Record terms | $R ::= \overrightarrow{N\;\mathsf{as}\;\ell}$ |
| Base terms | $X ::= x.\ell \mid c(\vec{X}) \mid \mathsf{empty}\;L$ |

Continuing our example, $\mathbf{L}(q_1)$ and $\mathbf{L}(q_2)$ translate to $q_1'$ and $q_2'$ where:

$q_1'$ = select $x$.name as i1_name,
    row_number() over (order by $x$.name) as i1_people
  from *departments* as $x$
$q_2'$ = (with $q$ as (select $x$.name as i1_name,
      row_number() over (order by $x$.name) as i2
    from *departments* as $x$)
  select $a$ as i1_1, $z$.i2 as i1_2, $y$.name as i2_name, $b$ as i2_tasks_1,
    row_number() over (order by $z$.i1_name, $z$.i2,
      $y$.dept, $y$.employee, $y$.salary) as i2_tasks_2
  from *employees* as $y$, $q$ as $z$
  where ($z$.i1_name $= y$.dept $\wedge$
      ($y$.salary $< 1000 \vee y$.salary $> 1000000$)))
 union all
 (with $q$ as (select $x$.name as i1_name,
      row_number() over (order by $x$.name) as i2
    from *departments* as $x$)
  select $a$ as i1_1, $z$.i2 as i1_2, $y$.name as i2_name, $d$ as i2_tasks_1,
    row_number() over (order by $z$.i1_name, $z$.i2,
      $y$.dept, $y$.name, $y$.client) as i2_tasks_2
  from *contacts* as $y$, $q$ as $z$
  where ($z$.i1_name $= y$.dept $\wedge y$.client))

Modulo record flattening, the above fragment of SQL is almost isomorphic to the image of the let-insertion translation. The only significant difference is the use of row_number in place of index. Each instance of index in the body $R$ of a subquery of the form for $(\overrightarrow{x \leftarrow t}$ where $X)$ return $R$ is simulated by a term of the form row_number() over (order by $\overrightarrow{x.\ell}$), where:

$$x_i : \langle \ell_{i,1} : A_{i,1}, \dots, \ell_{i,m_i} : A_{i,m_i}\rangle$$
$$\overrightarrow{x.\ell} = x_1.\ell_{1,1}, \dots, x_1.\ell_{1,m_1}, \quad \dots, \quad x_n.\ell_{n,1}, \dots, x_n.\ell_{n,m_n}$$

```
QF1: SELECT e.emp FROM employees e
     WHERE e.salary > 10000
QF2: SELECT e.emp, t.tsk
     FROM employees e, tasks t
     WHERE e.emp = t.emp
QF3: SELECT e1.emp, e2.emp
     FROM employees e1, employees e2
     WHERE e1.dpt = e2.dpt
       AND e1.salary = e2.salary
       AND e1.emp <> e2.emp
QF4: (SELECT t.emp FROM tasks t
      WHERE t.tsk = 'abstract')
     UNION ALL (SELECT e.emp FROM employees
                WHERE e.salary > 50000)
QF5: (SELECT t.emp FROM tasks t
      WHERE t.tsk = 'abstract')
     MINUS
     (SELECT e.emp FROM employees e
      WHERE e.salary > 50000)
QF6: ((SELECT t.emp FROM tasks t
       WHERE t.tsk = 'abstract')
      UNION ALL (SELECT e.emp FROM employees e
                 WHERE e.salary > 50000))
     MINUS
     ((SELECT t.emp FROM tasks t
       WHERE t.tsk = 'enthuse')
      UNION ALL (SELECT e.emp FROM employees e
                 WHERE e.salary > 10000))
```

**Figure 8: SQL versions of flat queries used in experiments**

A possible concern is that row_number is non-deterministic. It computes row numbers ordered by the supplied columns, but if there is a tie, then it is free to order the equivalent rows in any order. However, we always order by *all* columns of all tables referenced from the current subquery, so our use of row_number is always deterministic. (An alternative could be to use nonstandard features such as PostgreSQL's OID or MySQL's ROWNUM, but sorting would still be necessary to ensure consistency across inner and outer queries.)

## 8. EXPERIMENTAL EVALUATION

Ferry's loop-lifting translation has been implemented in Links previously by Ulrich, a member of the Ferry team [25], following the approach described by Grust et al. [11] to generate SQL:1999 algebra plans, then calling Pathfinder [13] to optimise and evaluating the resulting SQL on a PostgreSQL database. We have also implemented query shredding in Links, running against PostgreSQL; our implementation[1] does not use Pathfinder. We performed initial experiments with a larger ad hoc query benchmark, and developed some optimisations, including inlining certain `WITH` clauses to unblock rewrites, using keys for row numbering, and implementing stitching in one pass to avoid construction of intermediate in-memory data structures that are only used once and then discarded. We report on shredding with all of these optimisations enabled.

*Benchmark queries.* There is no standard benchmark for queries returning nested results. In particular, the popular TPC-H benchmark is not suitable for comparing shredding and loop-lifting: the TPC-H queries do not return nested results, and can be run directly on any SQL-compliant RDBMS, so neither approach needs to do any work to produce an SQL query.

We selected twelve queries over the organisation schema described in Section 3 to use as a benchmark. The first six queries,

[1]`http://github.com/slindley/links/tree/shredding`

```
Q1 : for (d ← departments)
     return (⟨name = d.name,
             employees = employeesOfDept d,
             contacts = contactsOfDept d⟩
Q2 : for (d ← Q1)
     where (all d.employees (λx.contains x.tasks "abstract"))
     return ⟨dept = d.dept⟩
Q3 : for (e ← employees)
     return ⟨name = e.name, task = tasksOfEmp(e)⟩
Q4 : for (d ← departments)
     return ⟨dept = d.dept, employees = for (e ← employees)
                                        where (d.dept = e.dept)
                                        return e.employee⟩
Q5 : for (t ← tasks) return ⟨a = t.task, b = employeesByTask t⟩
Q6 : for (d ← Q1)
     return (⟨department = d.name,
             people =
               getTasks(outliers(d.employees)) (λy. y.tasks)
             ⊎ getTasks(clients(d.contacts)) (λy. return "buy")⟩)
```

**Figure 9: Nested queries used in experiments**

named `QF1`–`QF6`, return flat results and can be translated to SQL using existing techniques, without requiring either shredding or loop-lifting. We considered these queries as a sanity check and in order to measure the overhead introduced by loop-lifting and shredding. Figure 8 shows the SQL versions of these queries.

We also selected six queries `Q1`–`Q6` that do involve nesting, either within query results or in intermediate stages. They are shown in Figure 9; they use the auxiliary functions defined in Section 3. `Q1` is the query $Q_{org}$ that builds the nested organisation view from Section 3. `Q2` is a flat query that computes a flat result from `Q1` consisting of all departments in which all employees can do the "abstract" task; it is a typical example of a query that employs higher-order functions. `Q3` returns records containing each employee and the list of tasks they can perform. `Q4` returns records containing departments and the set of employees in each department. `Q5` returns a record of tasks paired up with sets of employees and their departments. `Q6` is the outliers query $Q$ introduced in Section 3.

*Experimental results.* We measured the query execution time for all queries on randomly generated data, where we vary the number of departments in the organisation from 4 to 4096 (by powers of 2). Each department has on average 100 employees and each employee has 0–2 tasks, and the largest (4096 department) database was approximately 500MB. Although the test datasets are moderate in size, they suffice to illustrate the asymptotic trends in the comparative performance of the different techniques. All tests were performed using PostgreSQL 9.2 running on a MacBook Pro with 4-core 2.6GHz CPU, 8GB RAM and 500GB SSD storage, with the database server running on the same machine (hence, negligible network latency). We measure *total* time to translate a nested query to SQL, evaluate the resulting SQL queries, and stitch the results together to form a nested value, measured from Links.

We evaluated each query using query shredding and loop-lifting, and for the flat queries we also measured the time for Links' default (flat) query evaluation. The experimental results for the flat queries are shown in Figure 10 and for the nested queries in Figure 11. Note that both axes are logarithmic. All times are in milliseconds; the times are medians of 5 runs. The times for small data sizes provide a comparison of the overhead associated with shredding, loop-lifting or Links' default query normalisation algorithm.

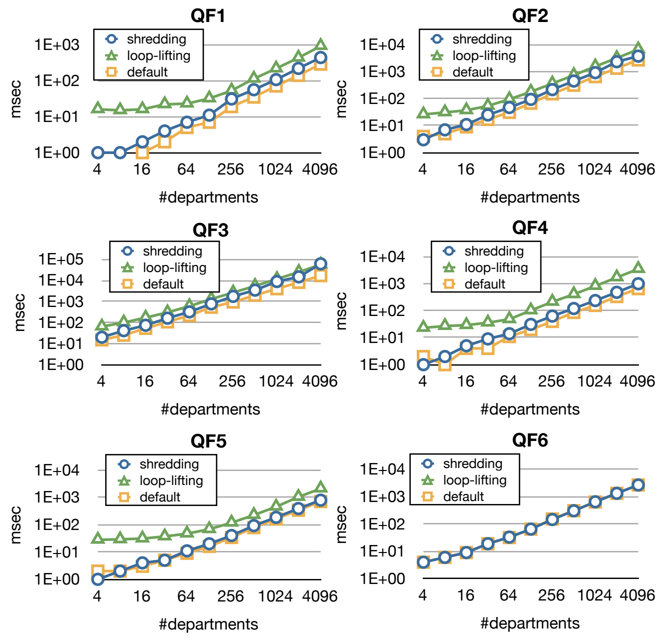

**Figure 10: Experimental results (flat queries)**

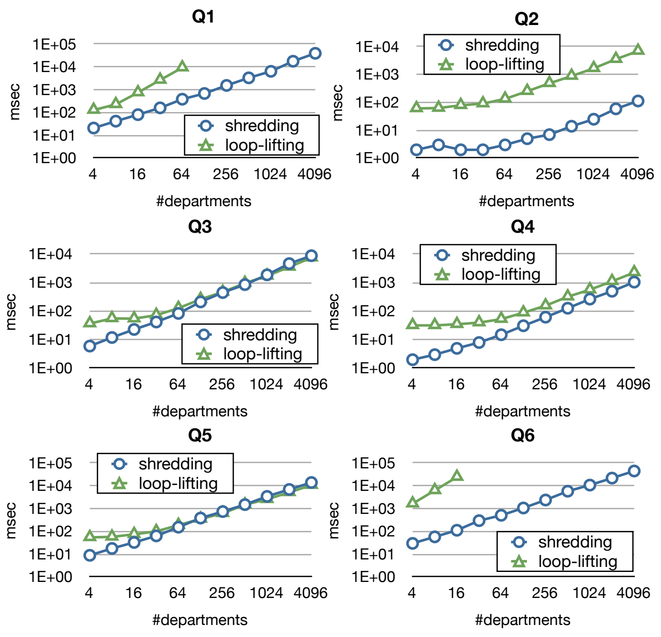

**Figure 11: Experimental results (nested queries)**

*Discussion.* We should re-emphasise that Ferry (and Ulrich's loop-lifting implementation for Links) supports grouping and aggregation features that are not handled by our translation. We focused on queries that both systems can handle, but Ferry has a clear advantage for grouping and aggregation queries or when ordering is important (e.g. sorting or top-$k$ queries). Ferry is based on list semantics, while our approach handles multiset semantics. So, some performance differences may be due to our exploitation of multiset-based optimisations that Ferry (by design) does not exploit.

The results for flat queries show that shredding has low per-query overhead in most cases compared to Links' default flat query evaluation, but the queries it generates are slightly slower. Loop-lifting, on the other hand, has a noticeable per-query overhead, likely due to its invocation of Pathfinder and associated serialisation costs. In some cases, such as `QF4` and `QF5`, loop-lifting is noticeably slower asymptotically; this appears to be due to additional sorting needed to maintain list semantics. We encountered a bug that prevented loop-lifting from running on query `QF6`; however, shredding had negligible overhead for this query. In any case, the overhead of either shredding or loop-lifting for flat queries is irrelevant: we can simply evaluate such queries using Links' default flat query evaluator. Nevertheless, these results show that the overhead of shredding for such queries is not large, suggesting that the queries it generates are similar to those currently generated by Links. (Manual inspection of the generated queries confirms this.)

The results for nested queries are more mixed. In most cases, the overhead of loop-lifting is dominant on small data sizes, which again suggests that shredding may be preferable for OLTP or Web workloads involving rapid, small queries. Loop-lifting scales poorly on two queries (`Q1` and `Q6`), and did not finish within 1 minute even for small data sizes. Both `Q1` and `Q6` involve 3 levels of nesting and in the innermost query, loop-lifting generates queries with Cartesian products inside OLAP operators such as `DENSE_RANK` or `ROW_NUMBER` that Pathfinder was not able to remove. The queries generated by shredding in these cases avoid this pathological behaviour. For other queries, such as `Q2` and `Q4`, loop-lifting performs better but is still slower than shredding as data size increases. Comparison of the queries generated by loop-lifting and shredding reveals that loop-lifting encountered similar problems with hard-to-optimise OLAP operations. Finally, for `Q3` and `Q5`, shredding is initially faster (due to the overhead of loop-lifting and calling Pathfinder) but as data size increases, loop-lifting wins out. Inspection of these generated queries reveals that the queries themselves are similar, but the shredded queries involve more data movement. Also, loop-lifting returns sorted results, so it avoids in-memory hashing or sorting while constructing the nested result. It should be possible to incorporate similar optimisations into shredding to obtain comparable performance.

Our experiments show that shredding performs similarly or better than loop-lifting on our (synthetic) benchmark queries on moderate (up to 500MB) databases. Further work may need to be done to investigate scalability to larger databases or consider more realistic query benchmarks.

## 9. RELATED AND FUTURE WORK

We have discussed related work on nested queries, Links, Ferry and LINQ in the introduction. Besides Cooper [7], several authors have recently considered higher-order query languages. Benedikt et al. [1, 27] study higher-order queries over flat relations. The Links approach was adapted to LINQ in F# by Cheney et al. [5]. Higher-order features are also being added to XQuery 3.0 [21].

Research on shredding XML data into relations and evaluating XML queries over such representations [17] is superficially similar to our work in using various indexing or numbering schemes to handle nested data. Grust et al.'s work on translating XQuery to SQL via Pathfinder [13] is a mature solution to this problem, and Grust et al. [12] discuss optimisations in the presence of unordered data processing in XQuery. However, XQuery's ordered data model and treatment of node identity would block transformations in our algorithm that assume unordered, pure operations.

We can now give a more detailed comparison of our approach with the indexing strategies in Van den Bussche's work and in

Ferry. Van den Bussche's approach uses natural indexes (that is, $n$-tuples of ids), but does not preserve multiset semantics. Our approach preserves multiplicity and can use natural indexes, we also propose a flat indexing scheme based on row_number. In Ferry's indexing scheme, the surrogate indexes only link adjacent nesting levels, whereas our indexes take information at all higher levels into account. Our flat indexing scheme relies on this property, and Ferry's approach does not seem to be an instance of ours (or vice versa). Ferry can generate multiple SQL:1999 operations and Pathfinder tries to merge them but cannot always do so. Our approach generates row_number operations only at the end, and does not rely on Pathfinder. Finally, our approach uses normalisation and tags parts of the query to disambiguate branches of unions.

Loop-lifting has been implemented in Links by Ulrich [25], and Grust and Ulrich [15] recently presented techniques for supporting higher-order functions as query results. By using Ferry's loop-lifting translation and Pathfinder, Ulrich's system also supports list semantics and aggregation and grouping operations; to our knowledge, it is an open problem to either prove their correctness or adapt these techniques to fit our approach. Ferry's approach supports a list-based semantics for queries, while we assume a bag-based semantics (matching SQL's default behaviour). Either approach can accommodate set-based semantics simply by eliminating duplicates in the final result. In fact, however, we believe the core query shredding translation (Sections 4–6) works just as well for a list semantics. The only parts that rely on unordered semantics are normalisation (Section 2.2) and conversion to SQL (Section 7). We leave these extensions to future work.

Our work is also partly inspired by work on unnesting for nested data parallelism. Blelloch and Sabot [2] give a compilation scheme for NESL, a data-parallel language with nested lists; Suciu and Tannen [23] give an alternative scheme for a nested list calculus. This work may provide an alternative (and parallelisable) implementation strategy for Ferry's list-based semantics [11].

## 10. CONCLUSION

Combining efficient database access with high-level programming abstractions is challenging in part because of the limitations of flat database queries. Currently, programmers must write flat queries and manually convert the results to a nested form. This damages declarativity and maintainability. Query shredding can help to bridge this gap. Although it is known from prior work that query shredding is possible in principle, and some implementations (such as Ferry) support this, getting the details right is tricky, and can lead to queries that are not optimised well by the relational engine. Our contribution is an alternative shredding translation that handles queries over bags and delays use of OLAP operations until the final stage. Our translation compares favourably to loop-lifting in performance, and should be straightforward to extend and to incorporate into other language-integrated query systems.

*Acknowledgements.* We thank Ezra Cooper, Torsten Grust, and Jan Van den Bussche for comments. We are very grateful to Alex Ulrich for sharing his implementation of Ferry in Links and for extensive discussions. This work was supported by EPSRC grants EP/J014591/1 and EP/K034413/1 and a Google Research Award.

## 11. REFERENCES

[1] M. Benedikt, G. Puppis, and H. Vu. Positive higher-order queries. In *PODS*, pages 27–38, 2010.

[2] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.*, 8:119–134, February 1990.

[3] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23, 1994.

[4] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.

[5] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ICFP*, pages 403–416. ACM, 2013.

[6] J. Cheney, S. Lindley, and P. Wadler. Query shredding: efficient relational evaluation of queries over nested multisets (extended version). Technical Report arXiv:1404.7078, arXiv.org, 2014. http://arxiv.org/abs/1404.7078.

[7] E. Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *DBPL*, pages 36–51, 2009.

[8] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: web programming without tiers. In *FMCO*, volume 4709 of *LNCS*, 2007.

[9] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25:457–516, December 2000.

[10] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: Database-supported program execution. In *SIGMOD*, June 2009.

[11] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-safe LINQ compilation. *PVLDB*, 3(1), 2010.

[12] T. Grust, J. Rittinger, and J. Teubner. eXrQuy: Order indifference in XQuery. In *ICDE*, pages 226–235, 2007.

[13] T. Grust, J. Rittinger, and J. Teubner. Pathfinder: XQuery off the relational shelf. *IEEE Data Eng. Bull.*, 31(4):7–14, 2008.

[14] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL hosts. In *VLDB*, pages 252–263, 2004.

[15] T. Grust and A. Ulrich. First-class functions for first-order database engines. In *DBPL*, 2013.

[16] W. Kim. On optimizing an SQL-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, Sept. 1982.

[17] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-SQL query translation literature: The state of the art and open problems. In *Xsym*, pages 1–18, 2003.

[18] S. Lindley and J. Cheney. Row-based effect types for database integration. In *TLDI*, 2012.

[19] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, 2006.

[20] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[21] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XQuery 3.0: An XML query language. W3C Proposed Recommendation, October 2013. http://www.w3.org/TR/xquery-30/.

[22] H. J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Inf. Syst.*, 11:137–147, April 1986.

[23] D. Suciu and V. Tannen. Efficient compilation of high-level data parallel algorithms. In *SPAA*, pages 57–66. ACM, 1994.

[24] D. Syme, A. Granicz, and A. Cisternino. *Expert F# 3.0*. Apress, 2012.

[25] A. Ulrich. A Ferry-based query backend for the Links programming language. Master's thesis, University of Tübingen, 2011. Code supplement: http://github.com/slindley/links/tree/sand.

[26] J. Van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theor. Comput. Sci.*, 254(1-2), 2001.

[27] H. Vu and M. Benedikt. Complexity of higher-order queries. In *ICDT*, pages 208–219. ACM, 2011.

[28] P. Wadler. Comprehending monads. *Math. Struct. in Comp. Sci.*, 2(4), 1992.

[29] L. Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, 52(3), 1996.

[30] L. Wong. Kleisli, a functional query system. *J. Funct. Programming*, 10(1), 2000.