The Flatter, the Better

Query Compilation Based on the Flattening Transformation

Alexander Ulrich Torsten Grust
Universität Tübingen
Tübingen, Germany
[alexander.ulrich, torsten.grust]@uni-tuebingen.de

ABSTRACT

We demonstrate the insides and outs of a query compiler based on the **flattening transformation**, a translation technique designed by the programming language community to derive efficient data-parallel implementations from iterative programs. Flattening admits the straightforward formulation of intricate query logic—including deeply nested loops over (possibly ordered) data or the construction of rich data structures. To demonstrate the level of expressiveness that can be achieved, we will bring a compiler frontend that accepts queries embedded into the Haskell programming language. Compilation via flattening takes places in a series of simple steps all of which will be made tangible by the demonstration. The final output is a program of lifted primitive operations which existing query engines can efficiently implement. We provide backends based on PostgreSQL and VectorWise to make this point—however, most set-oriented or data-parallel engines could benefit from a flattening-based query compiler.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*Query languages, Database (persistent) programming languages*

General Terms: Languages, Performance

Keywords: Nested data parallelism; flattening; list comprehensions

1. 20 YEARS OF FLATTENING

About 20 years ago, the **flattening transformation** has been devised to compile the NESL programming language into the data-parallel primitives of a vector machine [2]. Today, we demonstrate how flattening can be reinterpreted to compile expressive query languages into the bulk-oriented primitives of existing database engines. From NESL, we inherit (1) its ability to efficiently cope with iteration, even if nested deeply, (2) its support for nested and ordered data models, and (3) a functional and compositional semantics.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2735359.

In tandem with established query compilation techniques that we repeatedly draw on here, flattening enables a principled translation that proceeds in small digestible steps—a welcome advance over rather complex and monolithic approaches, including our own [7]. Flattening admits rich user-facing query languages and data models beyond those offered by recent related efforts (*query shredding* [4], for example, lacks support for ordered data, grouping, or aggregation). To make these points we express the examples in this paper in a comprehension-based language, *Database-Supported Haskell*[1], that is deeply embedded into Haskell [11]. The flattening idea is not tied to a particular frontend language, however. The core of flattening itself may be described by a compact set of transformation rules that trade nested iteration for a small library of—possibly lifted, see Section 3—query engine built-ins. Existing database backends can efficiently support these lifted built-ins. We will bring PostgreSQL [12] and Vectorwise [17] for demonstration, but it is a hypothesis of this work that a variety of database engines and data-parallel execution frameworks (*e.g.*, Stratosphere [1]) can be a driven by a flattening-based compiler.

The demonstration features a full-stack implementation of flattening, from the Haskell frontend to the database backend. All intermediary compilation artifacts are made tangible for the demo audience.

2. LOOPS, LISTS, AND LAYERS

Let us begin with a review of three sample queries, similar to those the demo audience will experience on site. The queries are formulated in Haskell, in a style that is idiomatic for programs that process lists. Flattening can compile query languages whose expressiveness and data model support comes close to those of general-purpose programming languages. We exploit this here to let the border between programs and queries vanish. (The details of this query embedding are orthogonal to flattening and have been described elsewhere [6].)

No knowledge of Haskell is required to grasp the present paper. The following should suffice: Principal query construct is the loop (or iteration), expressed in *list comprehension* syntax `[e | x <- xs, p]`, which iterates over the elements x of list xs. Expression e is evaluated for all x that pass predicate p (typically, x occurs free in e and p) to form the result list [3]. We write e `::` a to denote that expression e has type a. Programs use e `::` `Q` a to signal that they intend expression e to be evaluated by an un-

[1] `http://hackage.haskell.org/package/DSH`

trades

id	ts	day	price
ACME	1	10/20/2014	3.0
ACME	2	10/20/2014	4.0
ACME	3	10/20/2014	1.0
ACME	4	10/20/2014	7.0
⋮	⋮	⋮	⋮

Figure 1: Stock trading data. Rows have Haskell record type `Trade`. Fields of a row are accessed using selector functions: `price` x is `3.0` if x denotes the first row.

```
1  -- rolling minimum (mins [3,4,1,7] = [3,3,1,1])
2  mins :: Ord a => Q [a] -> Q [a]
3  mins xs =
4    [ minimum [ y | (y,j) <- #xs, j <= i ] | (_,i) <- #xs ]
5
6  -- margin ≙ current value - minimum value up to now
7  margins :: (Ord a, Num a) => Q [a] -> Q [a]
8  margins xs = [ x - y | (x,y) <- zip xs (mins xs) ]
9
10 -- our profit is the maximum margin obtainable
11 profit :: (Ord a, Num a) => Q [a] -> Q a
12 profit xs = maximum (margins xs)
13
14 -- best profit obtainable for stock on given date
15 bestProfit :: Text -> Date -> Q [Trade] -> Q Double
16 bestProfit stock date trades =
17   profit [ price t | t <- sortWith ts trades,
18                      id t  == toQ stock,
19                      day t == toQ date ]
```

Figure 2: Best profit obtainable if we buy, then sell stock. `bestProfit "ACME" "10/20/2014" trades` yields `6.0`.

derlying database system (*i.e.*, not by the regular Haskell runtime). Database tables are accessed like lists: for example, we use `trades :: Q [Trade]` to obtain the list of rows in table trades of Figure 1 in primary key order.

A list-based data model. With flattening, order is the default. Queries operate over lists (or: arrays, vectors) and use operations that preserve, depend on, or establish the order of list items.

Consider a trading application where a timestamp is used to record the price fluctuation of stocks over time (see column ts of table trades in Figure 1—this scenario is taken directly from [9]). What is the best possible profit if we buy and then sell ACME stocks on October 20? At timestamp t, our margin is ACME's current price minus its minimum price at or before t. This margin is what we try to maximize. The list-based query of Figure 2 implements this strategy almost literally. Main function `bestProfit` sorts the trading data in timestamp order, filters the stock and day of interest, and then calls on auxiliary functions (`profit`, `margins`, and `mins`) to process the list of stock prices.

Note how function `mins` computes a rolling minimum (*e.g.*, `mins [3.0,4.0,1.0,7.0] = [3.0,3.0,1.0,1.0]`) and thus is of general utility (indeed `mins` will work for lists of any type `a` as long as its values may be ordered: `Ord a`). Auxiliary functions `mins` and `margins` themselves invoke the primitive operation `#` (to make item order explicit, `#`$[x_1,\ldots,x_n]$ = `[(`x_1`,1),...,(`x_n`,n)]`) and `zip` (to pair the items of lists, `zip` $[x_1,\ldots,x_n]$ $[y_1,\ldots,y_n]$ = `[(`x_1`,`y_1`),...,(`x_n`,`y_n`)]`).

```
1 SELECT MAX(margins.price - margins.min)
2 FROM (SELECT t.price, MIN(t.price)
3                       OVER (ORDER BY t.ts
4                             ROWS BETWEEN UNBOUNDED PRECEDING
5                             AND CURRENT ROW)
6       FROM   trades AS t
7       WHERE  t.id = 'ACME'
8       AND    t.day = '10/20/2014') AS margins(price,min);
```

Figure 3: SQL:1999 code generated for the `bestProfit` query.

customer

c_custkey	c_name	c_nationkey	c_phone	c_acctbal
c	Customer#01	n	30-555-···	5236.89

lineitems

l_orderkey	l_partkey	l_extendedprice	l_discount
o	p	3712.08	0.05

nation

n_nationkey	n_name
n	USA

orders

o_orderkey	o_custkey	o_orderstatus	o_orderdate
o	c	P	10/20/14

Figure 4: Relevant excerpt of the TPC-H schema with sample rows. Matching key symbols (c, n, o) indicate relationships.

```
1  -- average account balance of the customers cs
2  avgBalance :: Q [Customer] -> Q Double
3  avgBalance cs = avg [ c_acctbal c | c <- cs, c_acctbal c > 0.0 ]
4
5  -- the orders of customer c (possibly none)
6  ordersOf :: Q Customer -> Q [Order]
7  ordersOf c = [ o | o <- orders, o_custkey o == c_custkey c ]
8
9  -- high potentials among the customers cs
10 potentialCustomers :: Q [Customer] -> Q [Customer]
11 potentialCustomers cs =
12   [ c | c <- cs, c_acctbal c > avgBalance cs, empty (ordersOf c) ]
13
14 -- country code (phone number prefix) for customer c
15 countryCodeOf :: Q Customer -> Q Text
16 countryCodeOf c = subString 2 (c_phone c)
17
18 -- does customer c live in any of the given countries?
19 livesIn :: Q Customer -> [Text] -> Q Bool
20 livesIn c countries = countryCodeOf c `elem` toQ countries
21
22 -- TPC-H query Q22
23 q22 :: [Text] -> Q [(Text, Integer, Double)]
24 q22 countries =
25   sortWith (\(country, _, _) -> country)
26     [ (country, length custs, sum (map c_acctbal custs)) |
27       (country, custs) <- groupWith countryCodeOf pots ]
28   where
29     pots = potentialCustomers [ c | c <- customers,
30                                     c `livesIn` countries ]
```

Figure 5: A Haskell formulation of TPC-H query *Q22*.

Flattening plus subsequent code generation for PostgreSQL can translate this Haskell query into the SQL:1999 code of Figure 3. We argue that this generated SQL formulation is reasonable in that it closely resembles the hand-written solution to the best profit problem proposed in [9]. The semantics of timestamp ordering, in particular, do not infect the entire generated code but remain local to the ordered `MIN` window aggregate (lines 2 to 5).

Layered queries (primitive, generic, domain-specific). Dividing queries into cooperating, tiny (typically single-line) auxiliary functions can help to (1) formulate complex query logic and (2) identify reusable, generic query pieces. Once we adopt this style, well-known scenarios like the TPC-H database (see the excerpt in Figure 4) and its associated queries appear in a new light.

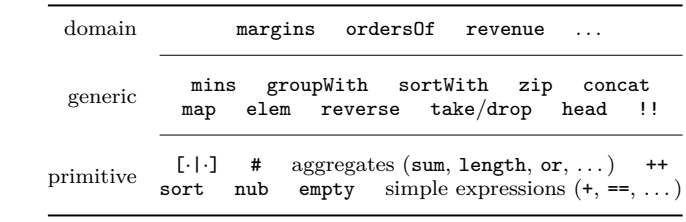


Figure 6: Three layers of query abstractions. Operations in upper layers are definable in terms of those on lower layers.

```
1  SELECT substring(c.c_phone, 1, 2), COUNT(*), SUM(c.c_acctbal)
2  FROM customer AS c,
3       (SELECT AVG(c1.c_acctbal)
4        FROM customer AS c1
5        WHERE substring(c1.c_phone, 1, 2) IN (VALUES ('44'),('49'))
6          AND c1.c_acctbal > 0.0) AS avgBalance(c_acctbal)
7  WHERE substring(c.c_phone, 1, 2) IN (VALUES ('44','49')
8        AND c.c_custkey NOT IN (SELECT o.o_custkey
9                                FROM   orders AS o)
10       AND c.c_acctbal > avgBalance.c_acctbal
11 GROUP BY substring(c.c_phone, 1, 2)
12 ORDER BY substring(c.c_phone, 1, 2);
```

Figure 7: PostgreSQL-executable code for `q22 ["44","49"]`.

Given a list of countries, TPC-H query *Q22* [16] aims to identify potential customers: those whose accounts show an above-average balance but currently have not placed any order. Average account balance or current order placement are pervasive concepts in the TPC-H application domain which leads us to define designated functions (see `avgBalance` and `ordersOf` in Figure 5—`ordersOf`, in particular, embodies the *1:n* relationship between customers and orders). With these domain-specific functions in place, the core of the problem becomes a two-liner itself (lines 11 and 12). The residual query below line 15 filters, groups, and sorts customers (by phone country code) to prepare the answer format that *Q22* demands.

More generally, we obtain a multi-layered query language design (Figure 6) in which a small set of primitives are used to define generic, domain-agnostic query operations. The polymorphic types of the functions in this second layer (cf. the rolling minimum `mins`) hint at their reusability. Domain-specific functions in the top layer embody application-level concepts and can lead to very succinct query formulation ("one-liners").

It is particularly desirable to populate the top layers and keep the set of primitives as small as possible (in *AQuery* [9], for example, the equivalent of `mins` lives at the bottom layer). Flattening is fully compositional and makes the required unrestricted composition of query abstractions viable. Users are not bound to query templates of any form and are not penalized for the introduction or use of abstractions. Figure 7 shows the generated (monolithic) SQL code—the used abstractions have been compiled away.

Nested data: separation of contents and structure. Query languages whose data modeling capabilities are on par with programming languages are the exception rather than the rule [5]. With its roots in the programming language domain, flattening can compile queries that arbitrarily nest two principal type constructors—tuples `(·,·)` and lists `[·]`.

Nested data of this kind allows the construction of complex reports: function `expectedRevenue` (see Figure 8) pairs each customer with the list of dates and expected revenues for her pending orders. True to layering, most of the query logic has been relocated into domain-specific functions to control complexity and promote reusability: while auxiliary function `revenue` embodies TPC-H's specific notion of order revenue, function `ordersWithStatus` builds on `ordersOf` (see above) to find all orders in condition `status` for customer `c`. The overall result type has shape `[(·,[(·,·)])]`.

```
1  -- is customer c a resident of nation?
2  hasNationality :: Q Customer -> Text -> Q Bool
3  hasNationality c nation =
4    or [ n_name n == toQ nation && n_nationkey n == c_nationkey c |
5         n <- nations ]
6
7  -- all orders of customer c with the given status (O, P, C)
8  ordersWithStatus :: Text -> Q Customer -> Q [Order]
9  ordersWithStatus status c =
10   [ o | o <- ordersOf c, o_orderstatus o == toQ status ]
11
12 -- our revenue for order o
13 revenue :: Q Order -> Q Double
14 revenue o = sum [ l_extendedprice l * (1 - l_discount l) |
15                   l <- lineitems, l_orderkey l == o_orderkey o ]
16
17 -- expected revenues (by customer, with details) in nation
18 expectedRevenue :: Text -> Q [(Text, [(Date, Double)])]
19 expectedRevenue nation =
20   [ (c_name c, [ (o_orderdate o, revenue o) |
21                  o <- ordersWithStatus "P" c ]) |
22     c <- customers,
23     c `hasNationality` nation ]
```

Figure 8: Expected revenue report (produces nested result).

Efficient support for such nested data structures poses a challenge for common database backend architectures with their rigid layout of data (*e.g.*, first normal form relational tables). Flattening follows a compilation scheme that separates data contents from any nesting structure. The separation and the subsequent re-imprinting of structure are essentially compile-time operations (Section 3). This scheme turns out to be a good fit for existing database backends which then can compute contents and structural aspects separately: the demo audience will understand how a query of list nesting depth d will yield d backend queries ($d = 2$ for the `expectedRevenue` example above).

3. 10 MINUTES OF FLATTENING

It is the prime purpose of this demonstration to provide a comprehensible and concise account of the role that flattening can play in query compilation. Following the *compilation by transformation* principle, the surface query syntax is lowered towards database-executable form in a series of steps, each of which yield human-readable and self-contained intermediate output. Here, we sketch the individual steps. The demo audience should be able to follow a complete walkthrough in about 10 minutes. Figure 9 provides a road map.

① **Desugar and normalize.** As a preparatory step, the compiler removes domain-specific and generic query abstractions, replacing them with their primitive equivalents (cf. Figure 6). The bodies of user-defined functions like `margins` or `revenue` are unfolded at their call sites. (This implies that these functions must be non-recursive—primitives may embody recursive computation, however.) A polymorphic function in the generic layer is replaced by either

- its equivalent primitive comprehension form (refer to Figure 10) or

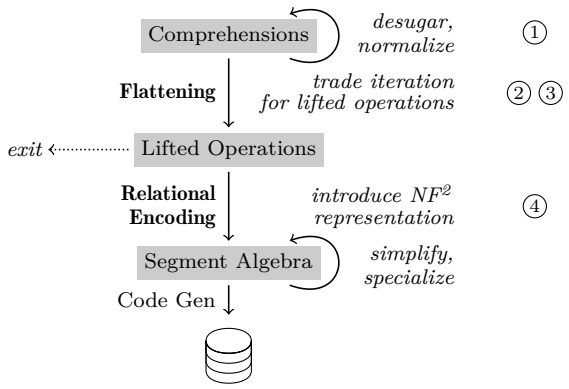


Figure 9: Compilation stages. Connect alternative backends at *exit* to receive the post-flattening input query.

generic	primitive
`groupWith f xs`	`[ (x,[ y | y <- xs, f y == x ]) | x <- nub (map f xs) ]`
`zip xs ys`	`[ (x,y) | (x,i) <- #xs, (y,j) <- #ys, i == j ]`
`concat xss`	`[ x | xs <- xss, x <- xs ]`
`take n xs`	`[ x | (x,i) <- #xs, i <= n ]`

Figure 10: Expressing generic query abstractions in terms of the primitive layer (excerpt).

- a special-purpose built-in primitive if that is provided by the underlying query engine (*e.g.*, `zip` may simply turn into the built-in `ZIP` if the engine features positional join support [10]).

After these replacements, the query of Figure 8 takes the form shown in Figure 11, for example.

Desugaring continues with normalizing rewrites that build on the extensive body of work on the unnesting and optimization of calculus-based or comprehension-style query languages [8, 15]: common comprehension patterns are traded for built-in operations which embody these patterns. Notably, this introduces list-based variants of joins, for example, `NESTJOIN` or `SEMIJOIN`.

② **Flattening: syntactic transformation.** All remaining comprehensions are in the normal form `[` e `|` x `<-` xs `]` (single generator, no predicates). Flattening, originally described by Blelloch [2] as well as Prins and Palmer [13], addresses the challenge of *nested data parallelism*: how to efficiently compile iterative programs of the given normal form if expression e itself contains iterative constructs. Typical for query-style programs, all examples in this paper manifest such nested iteration (up to depth 3, despite their simplicity).

Flattening revolves around the concept of *lifted functions* which consume and return—possibly in parallel—entire lists of items: the lifted variant of `F` $:: a \to b$ is `F`^{1} $:: [a] \to [b]$. (In the following, we identify `F` and `F`^{0}.) This notion of lifting coincides with the bulk-oriented mode of processing in query engines which are tuned to efficiently apply operations to entire collections of data items (*e.g.*, tables of rows). Flattening is specified via few syntactic program transformation rules. The gist is the following rule that expresses the iterated application of `F` in terms of its lifted variant:

$$\texttt{[F}^n\ e\ \texttt{|}\ x\ \texttt{<-}\ xs\ \texttt{]} \quad \mapsto \quad \texttt{F}^{n+1}\ \texttt{[}\ e\ \texttt{|}\ x\ \texttt{<-}\ xs\ \texttt{]} \qquad (\uparrow)$$

```
1  [ (c.c_name,
2     [ (o.o_orderdate,
3        SUM [ l.l_extendedprice * (1.0 - l.l_discount) |
4              l <- TABLE(lineitem),
5              l.l_orderkey == o.o_orderkey ]) |
6       o <- TABLE(orders),
7       o.o_custkey == c.c_custkey,
8       o.o_orderstatus == "P" ]) |
9    c <- TABLE(customer),
10   OR [ n.n_name == "USA" && c.c_nationkey == n.n_nationkey |
11        n <- TABLE(nation) ]
12 ]
```

Figure 11: The query of Figure 8 after domain-specific and generic abstractions have been unfolded. Comprehensions and primitives remain.

```
1 -- scan xs and identify spots (= True) where new maximum reached
2 climb :: Ord a => Q [a] -> Q [Bool]
3 climb xs = [ and [ y <= x | (y,j) <- #xs, j <= i ] | (x,i) <- #xs ]
4
5 -- does xs ascend monotonically?
6 monotonic :: Ord a => Q [a] -> Q Bool
7 monotonic xs = and (climb xs)
```

Figure 12: Function `climb xs`: identify where `xs` reaches new maxima (`climb [3,4,1,7] = [True,True,False,True]`).

Nested iteration yields liftings of order $n > 1$. For example, we obtain `[ [ F` y `|` y `<-` `G` x `] |` x `<-` xs `]` $\mapsto$ `F`^{2}`(G`^{1} xs`)` through two applications of ($\uparrow$) and the identity `[` x `|` x `<-` xs `]` $\equiv xs$. Since `F` $:: a \to b$, consequently `F`^{2} $:: [[a]] \to [[b]]$.

To provide a concise account of flattening, we trace the transformation of the one-liner `climb` function of Figure 12 (`climb`, quite similar to `mins` of Figure 2, and `monotonic` test whether the elements of a list grow monotonically). The transformation trace itself is found in Figure 13. After desugaring and normalization (step ①), the body of `climb` exhibits nested iteration:

$$\texttt{[}\ \cdots\ \texttt{[y <= x | (y,j) <- g] | (x,g) <-}\ \cdots\ \texttt{]}$$

The intermediate program after flattening (step ②) features the characteristic lifted built-ins (here: `AND`^{1}, `<=`^{2}, and pair projections like `.2`^{1} or `.1`^{2}). Explicit iteration has been compiled away entirely.

③ **Flattening: reducing `F`^{n} to `F`^{1}.** Flattening becomes practicable with the insight that any lifted built-in `F`^{n} with $n \geqslant 2$

```
  [ and [ y <= x | (y,j) <- #xs, j <= i ] | (x,i) <- #xs ]
=     ① (desugar, normalize)
  [ AND [ y <= x | (y,j) <- g ] |
     ((x,i),g) <- NESTJOIN{Ⓛ.2 <= Ⓡ.2} #xs #xs ]
=     ② (flatten)
  let xg = NESTJOIN{Ⓛ.2 <= Ⓡ.2} #xs #xs
   in AND¹ (xg.2¹.1² <=² (DIST¹ xg.1¹.1¹ xg.2¹))
=     ③ (reduce F^n to F¹)
  let xg = NESTJOIN{Ⓛ.2 <= Ⓡ.2} #xs #xs
      y = IMPRINT₁ xg.2¹ (FORGET₁ xg.2¹).1¹
   in AND¹ (IMPRINT₁ y
      ((FORGET₁ y) <=¹ (FORGET₁ (DIST¹ xg.1¹.1¹ xg.2¹))))
```

Figure 13: Tracing the flattening of `climb` (Figure 12). Ⓛ/Ⓡ inside the predicate `{`$\cdots$`}` of a built-in operation refer to its left/right input, `.1`/`.2` access components of a pair.

lifted	$\texttt{F}^n$ ($n \geqslant 2$, if $\texttt{F}^1$ is a built-in)
built-in	$\texttt{SEMIJOIN}^{0,1}$ $\texttt{NESTJOIN}^{0,1}$ $\texttt{RESTRICT}^{0,1}$ ... $\texttt{DIST}^{0,1}$ TABLE $\texttt{FORGET}_n$ $\texttt{IMPRINT}_n$ $\texttt{SORT}^{0,1}$ simple operators ($+^{0,1}$, $<=^{0,1}$, ...)

Figure 14: The underlying query engine is expected to provide the operations $\texttt{F}^0$ and $\texttt{F}^1$. Flattening reduces lifted operations $\texttt{F}^n$ (with $n \geqslant 2$) to these built-ins.

can be reduced to its variant $\texttt{F}^1$. This reduction (1) temporarily *forgets* about the $n-1$ outer nesting layers of $\texttt{F}^n$'s list argument *xs*, (2) applies $\texttt{F}^1$ to the—now presumed flat—argument, and then (3) *imprints* the outer list nesting layers on the result again:

$$\texttt{F}^n\ xs \quad \mapsto \quad \texttt{IMPRINT}_{n-1}\ xs\,(\texttt{F}^1\,(\texttt{FORGET}_{n-1}\ xs)) \qquad (\flat)$$

Query engines thus only need to implement the $\texttt{F}^0$ and $\texttt{F}^1$ variants. We have already said that $\texttt{F}^1$ fits well with the bulk-oriented execution model of these engines—this is obvious for simple operators like $\texttt{F} = +$ or $\texttt{F} = <=$ but extends to more complex built-ins like SEMIJOIN (our discussion of step ④ below touches on this). Figure 14 summarizes our expectations of which operations an underlying engine has to supply.

The efficiency of flattening–generated code hinges on the operations $\texttt{FORGET}_n$ and $\texttt{IMPRINT}_n$ to have *zero* run-time costs for any $n \geqslant 1$. Careful representations of nested data can provide this behavior (see below).

The end of step ③ marks the point where code generators for various query engines could be hooked up (cf. the *exit* in Figure 9). These engines will find opportunities for a bulk-oriented or data-parallel evaluation already made explicit through the lifting superscripts.

④ **Relational encoding.** Vanilla relational database systems are viable engines in the above sense. The zero run-time cost requirement for $\texttt{FORGET}_n$ and $\texttt{IMPRINT}_n$ suggest a data representation that *separates contents from structure*: given such a separation, $\texttt{FORGET}_n\ xs$ simply ignores the structure part of *xs* temporarily while $\texttt{IMPRINT}_n\ xs\ e$ applies (the already existing) structure of *xs* to *e*.

One relational representation of nested data that provides this separation is the trusted NF^2 model [14]. Lists of items are encoded in a ternary seg|pos|item contents table in which column pos keeps track of item order and column seg indicates the sub-list to which an item belongs. The order of these sub-lists is held in a *separate* seg|pos structure table referencing the contents table. A list nested to depth n will feature $n-1$ structure tables. Figure 15 visualizes how $\texttt{FORGET}_n/\texttt{IMPRINT}_n$ exploit this NF^2-style encoding of data: during the evaluation of the underlined subexpression of Figure 13, only $<=^1$ will incur run-time cost.

Relational engines already implement the operators $\texttt{F}^0$ (cf. Figure 14 again). For operators with atomic arguments and result ($\texttt{F} \in \{+, <=, \ldots\}$), $\texttt{F}^1$ is readily obtained through projection (π, mapping). At least two options exist to obtain lifted variants of bulk-oriented, or algebraic, built-ins F:

(1) extend the engine to provide true data-parallel implementations of $\texttt{F}^1$—note that all $\texttt{F}^1$ are intrinsically "embarrassingly parallel"—or

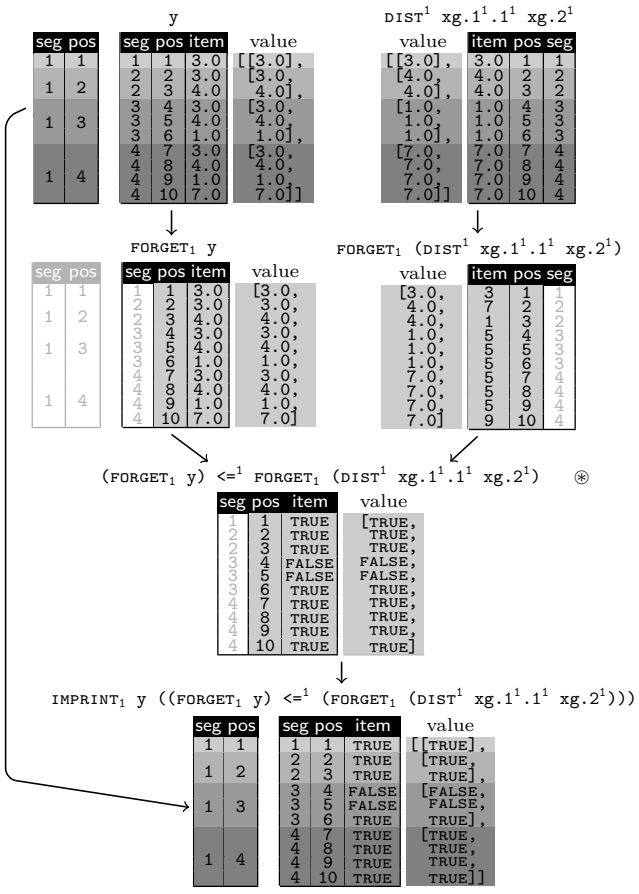


Figure 15: Evaluating the underlined expression in Figure 13. Operator $<=^1$ at ⊛ incurs the only run-time cost, $\texttt{FORGET}_1$ merely ignores (the grayed out) parts of the NF^2 encoding.

(2) parameterize the existing built-ins F to acknowledge list segmentation (column seg): $\texttt{SEMIJOIN}^1\{p\}$ then becomes $\ltimes\{(\text{L}).\mathsf{seg} = (\text{R}).\mathsf{seg} \wedge p\}$ and seg is prepended to the list of sort criteria in a SORT, for example.

The present work features a full implementation of option (2).

4. DEMONSTRATION SETUP

We will bring a demonstration setup that has been designed to facilitate both quick and deep impressions of flattening as a query compilation technique. A set of canned queries—similar to those discussed in this write-up— helps to provide an immediate overview of the flattening idea.

The demonstration does *not* run on rails, though: any canned query is editable, for example, to explore interesting edge cases in compilation. New ad hoc queries may be formulated. An interactive read-eval-print loop (REPL) promotes experimentation with queries, gives immediate feedback, and thus allows for quick "one-shot"demonstrations. Figure 16 shows the REPL (bottom of screenshot). Errors are signaled early: the system implements a static typing discipline that rejects programs for which no type assignment of the form $\texttt{Q}\,a$ can be found (these programs would not be database-executable).

The underlying relational backends, PostgreSQL and Vectorwise [17], will be preloaded with

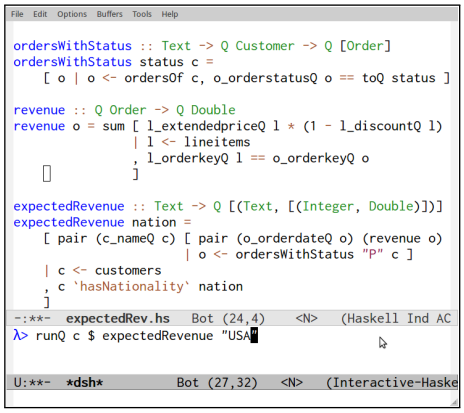


Figure 16: Haskell-embedded queries may be authored ad hoc and are evaluated in an interactive read-eval-print loop.

(1) toy data sets—that permit to explore query semantics since results may be checked item-by-item—as well as
(2) larger database instances so that the audience can assess backend performance once code has been generated.

We include familiar instances (*e.g.*, TPC-H) to ensure that attendees can easily follow the demonstration.

Going deeper. Along with the compiler implementation itself, the demonstration provides dedicated pretty-printers and visualizers for all intermediary program forms (screenshots in Figure 17). The inspection of the query after flattening (but before relational encoding) is of particular interest if new backends are to be connected. A peek at the code generated for the existing PostgreSQL and Vectorwise backends—SQL:1999 statements and algebraic plans, respectively—is provided as well. We hope to illustrate that a suitable encoding of segmented lists indeed (1) provides zero-cost implementations of $\texttt{FORGET}_n/\texttt{IMPRINT}_n$, and (2) leads to idiomatic relational queries that are not occupied with costly row order management.

5. REFERENCES

[1] A. Alexandrov et al. The Stratosphere Platform for Big Data Analytics. *VLDB Journal*, 2014.

[2] G. E. Blelloch and G. W. Sabot. Compiling Collection-Oriented Languages onto Massively Parallel Computers. *J. Parallel Distrib. Comput.*, 8(2), 1989.

[3] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *ACM SIGMOD Record*, 23(1), 1994.

[4] J. Cheney, S. Lindley, and P. Wadler. Query Shredding: Efficient Relational Evaluation of Queries Over Nested Multisets. In *Proc. SIGMOD*, 2014.

[5] G. Copeland and D. Maier. Making Smalltalk a Database System. In *Proc. SIGMOD*, 1984.

[6] G. Giorgidze, T. Grust, T. Schreiber, and J. Weijers. Haskell Boards the Ferry. In *Proc. IFL*, 2011.

[7] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-Safe LINQ Compilation. In *Proc. VLDB*, 2010.

[8] T. Grust and M. H. Scholl. How to Comprehend Queries Functionally. *J. Intell. Inf. Syst.*, 12(2-3), 1999.

[9] A. Lerner and D. Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *Proc. VLDB*, 2003.

[10] S. Manegold, P. Boncz, N. Nes, and Martin Kersten. Cache-Conscious Radix-Decluster Projections. In *Proc. VLDB*, 2004.

[11] S. Marlow. *Haskell 2010: Language Report*, 2010. `haskell.org`.

[12] *The PostgreSQL Relational Database System.* `postgresql.org`.

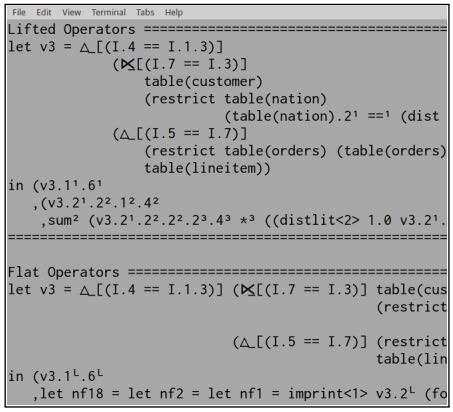
[13] J. F. Prins and D. W. Palmer. Transforming High-Level Data-Parallel Programs into Vector Operations. In *Proc. PPOPP*, 1993.

[14] H.-J. Schek and M. H. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 2(11), 1986.

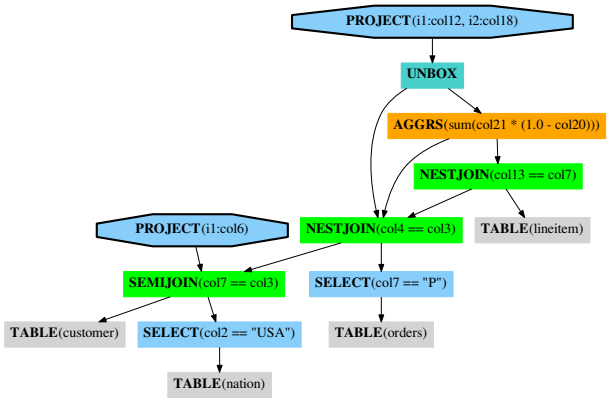
[15] H. J. Steenhagen, R. A. de By, and H. M. Blanken. Translating OSQL Queries into Efficient Set Expressions. In *Proc. EDBT*, 1996.

[16] *TPC Benchmark H (Rev. 2.17.0)*, 2013. `tpc.org/tpch`.

[17] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A Vectorized Analytical DBMS. In *Proc. ICDE*, 2012.



(a) Pretty-printed form after flattening and introduction of $\texttt{FORGET}_n/\texttt{IMPRINT}_n$ (compilation steps ② and ③).



(b) Diagram of plan immediately before code generation. Two root nodes (⬭) represent a result of depth $d = 2$.

Figure 17: Compilation artifacts are tangible for the demonstration audience (here: program `expectedRevenue` of Figure 8).