A Deep Embedding of Queries into Ruby

Torsten Grust [1], Manuel Mayr [2]

Department of Computer Science, Universität Tübingen
Tübingen, Germany
[1] torsten.grust@uni-tuebingen.de
[2] manuel.mayr@uni-tuebingen.de

***Abstract*—We demonstrate SWITCH, a deep embedding of relational queries into RUBY and RUBY on RAILS. With SWITCH, there is no syntactic or stylistic difference between RUBY programs that operate over in-memory array objects or database-resident tables, even if these programs rely on array order or nesting. SWITCH's built-in compiler and SQL code generator guarantee to emit few queries, addressing long-standing performance problems that trace back to RAILS' ACTIVERECORD database binding. "Looks likes RUBY, but performs like handcrafted SQL," is the ideal that drives the research and development effort behind SWITCH.**

I. RUBY ON RUTTED RAILS

RUBY on RAILS (or RAILS, for short) is found among the most actively deployed frameworks that support the rapid development of Web 2.0–style applications [11]. Notable RAILS applications abound, with *GitHub*, *Qype*, *Twitter*, or *Xing* being only a few of those widely recognized on the Web.

A RAILS–built application follows a strict *model–view–controller* design pattern [6]. To this end, the RAILS framework provides *domain-specific languages* (DSLs)—all embedded into the common host language RUBY—that enable developers to

- specify the application's call interface, typically in terms of a REST-ful URL router, [*controller*]
- design the client-facing presentation and interaction, typically based on HTML5/CSS/AJAX, and [*view*]
- to interface with a relational back-end that serves application data and holds state information. [*model*]

This demonstration focuses on RAILS' model DSL known as ACTIVERECORD. We refer to recent variants used from RAILS version 3 onwards, in which ACTIVERECORD is now based on ACTIVERELATION [1]. We argue that

(1) ACTIVERECORD's integration into the RUBY syntax, semantics, and data model, as well as
(2) the runtime interaction of a RAILS application and its relational database back-end

leave much to be desired. We will demonstrate a new design, SWITCH, that significantly improves on both aspects. RUBY's potential as a host language for database application development has been identified [10]. We believe that SWITCH can help to provide an intuitive and efficient environment for data-intensive programming in RUBY, also for developers who cannot or do not want to "drop into SQL." [2]

The subsequent discussion and the software demonstration itself will revolve around *Spree*, a versatile RAILS framework for the construction of Web shops [12]. *Spree* application data resides in relational tables whose layout closely resembles the TPC-H benchmark (Figure 1 shows an excerpt).

A Critique of ACTIVERECORD. Under the regime of ACTIVERECORD, queries are constructed through the chained invocation of specific *query methods* that operate on and return objects of class `ActiveRecord::Relation`, RAILS' primary query abstraction. These method invocation chains originate in table objects that directly reflect the base tables present in the underlying database back-end.[1]

"*What would be the cost of granting a discount on the open orders of all high-volume costumers?*" The code of Figure 2a is a typical RUBY snippet that answers this question using ACTIVERECORD's approach to query embedding. Figure 2b shows the SQL statements that ACTIVERECORD will derive from the RUBY snippet. While workable, a number of severe issues arise:

(1) The RUBY code is sprinkled with literal SQL text fragments (in quotes "···"): the blue fragments in Figure 2b are copied verbatim from the RUBY source. RUBY values are spliced in using parameters markers (`?`, `:tc`, `:s`). Such code is vulnerable to SQL injection attacks [13].
(2) Query methods—like `group`, `select`, `where`—implement a concatenative semantics of query construction: an `ActiveRecord::Relation` object simply gathers query clauses. Consumption of the query result, *e.g.* through the invocation of `map` (see lines 9 and 11 in Figure 2a), triggers an attempt to construct executable SQL text from the clauses gathered so far.

Note how (1) and (2) may easily lead to nonsense queries that fail at runtime. Further:

(3) Query construction is not as compositional as the SQL semantics: ACTIVERECORD often constructs sequences of separate SQL statements and uses the RUBY heap to carry sizable intermediate from query to query (see

[1] Following a RAILS convention, a RUBY table object named ⟨`T`⟩ (singular) is instantiated for any base table named ⟨`T`⟩`s` (plural). Looking at Figure 1, the RAILS application will refer to table objects `Order`, `Line_Item`, *etc.*

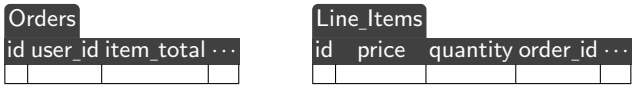


Fig. 1. Tables holding *Spree* application data (excerpt). The id columns serve as primary keys, order_id references Orders.

```ruby
discount = 20.0/100  # grant a 20% discount ...
high_vol = 10        # ... to customers with more than 10 open orders

high_vols = Order.group("user_id")
                 .having(["COUNT(user_id) >= ?", high_vol])
                 .select("user_id")

open_orders = Order.where(["user_id IN (:tc) AND state = :s",
                           { tc: high_vols.map(&:user_id),
                             s:  "O" }])
items = open_orders.includes(:Line_Item).map(&:line_items).flatten

cost = items.sum {|i| i.price * i.quantity} * discount
```

(a) RUBY code written in ACTIVERECORD–style. The red code fragments are not considered to be database-executable.

```
SELECT user_id
  FROM Orders
 GROUP BY user_id
HAVING COUNT(user_id) >= 10;

SELECT *
  FROM Orders
 WHERE user_id IN (4,7,...,1498,1499)
   AND state = 'O';

SELECT *
  FROM Line_Items
 WHERE order_id IN (1,2,...,59973,59974);
```

714 user ids

6124 order ids

(b) SQL statement sequence generated for the RUBY snippet of (a). Blue code is copied verbatim.

Fig. 2. ACTIVERECORD variant of the "discounting open orders" program. Intermediate results are carried from SQL statement to SQL statement via the RUBY heap (714 + 6124 integers are copied for a TPC-H instance of scale factor 0.01).

```ruby
discount = 20.0/100
high_vol = 10

high_vols = Order.group_by(&:user_id)
                 .select {|u,os| os.length >= high_vol}
open_orders = high_vols.map {|u,os| os.select {|o| o.state == "O"}}
                       .flatten
items = open_orders.map {|o| line_item.in_order(o)}.flatten

cost = items.sum {|i| i.price * i.quantity} * discount
```

(a) RUBY code written in array-processing style. The entire code fragment is considered database-executable by SWITCH.

```
SELECT SUM(li.price * li.quantity) * 0.2
  FROM  (SELECT o2.user_id, COUNT(*) AS cnt
           FROM Orders AS o2
       GROUP BY o2.user_id) AS oc,
       Orders AS o1,
       Line_Items AS li
 WHERE o1.id      = li.order_id
   AND o1.state   = 'O'
   AND oc.user_id = o1.user_id
   AND oc.cnt     > 10;
```

(b) SWITCH–generated SQL statement for the code snippet of (a). [Formatted for readability.]

Fig. 3. SWITCH's deep query embedding admits a natural comprehension-style formulation of the program of Figure 2a.

the arrows ⤳ and the resulting huge `IN (···)` clauses in Figure 2b which may easily overwhelm the back-end's SQL parser).

(4) Query results are post-processed by the RUBY interpreter although the relational back-end would be capable of performing the entire computation close to the source data. In Figure 2a, red code fragments are evaluated by RUBY instead of the database query processor.

II. SWITCH

SWITCH aims to provide a significantly more natural embedding of relational queries, leading RUBY onto new rails. We derive an expression tree representing the source RUBY program while the program executes. This tree is compiled into SQL statement text, executed on the database back-end, and the resulting table materialized on the RUBY heap in terms of an array of hashes. While we obviously cannot unroll all SWITCH design details here, we highlight a few defining features in this section.

A. Embedding Queries into Ruby

RUBY's consequent adoption of an open object model and its lenient syntactic conventions in which even basic control constructs are subject to runtime inspection and modification, make the language an ideal host for embedded DSLs. One recent example in the database arena is BLOOM, a RUBY-embedded DSL for distributed programming [3].

Deep Embedding. SWITCH realizes a *deep embedding* [8] of relational queries: there is *no* syntactic or stylistic difference between RUBY programs that operate over in-memory array objects or database-resident tables. SWITCH captures the program's structure in preparation for query generation. Developers continue to use the host language's versatile family of array operations (found in the RUBY modules `Array` and `Enumerable`). The set of SWITCH–supported operations includes

`map select group_by sort_by partition uniq flatten flat_map zip all? any? none? empty? one? member? append reverse take drop take_while drop_while at first last length avg sum max min max_by min_by`,

as well as arithmetics, comparisons, and Boolean operations. SWITCH respects array order and supports computation over nested arrays (which naturally arise with `group_by` or `partition`, for example). SWITCH's coverage of operations goes significantly beyond other RUBY database bindings, like DATAMAPPER, SEQUEL, or AMBITION [5].

Taking advantage of SWITCH's deep query embedding, Figure 3a shows the array-centric reformulation of the program of Figure 2a. Vanilla RUBY block syntax `{|x|···}` may now be used to specify operation arguments. Pattern matching, as used in the block expression `{|u,os| os.length >= high_vol}` (line 5), handily names and accesses the fields of a record—note that the second field `os` represents a nested `Array` object (here: an array of orders). RUBY idioms, *e.g.* shorthands

```
1 SELECT SUM(li.price * li.quantity) * (20.0/100) AS cost
2   FROM Orders AS o1,
3        Line_Items AS li
4  WHERE o1.id = li.order_id
5    AND o1.user_id IN (SELECT o2.user_id
6                         FROM Orders AS o2
7                       GROUP BY o2.user_id
8                         HAVING COUNT(*) > 10)
9    AND o1.state = 'O';
```

Fig. 4. "Discounting open orders": handcrafted SQL text (also see column SQL (✎) in **?? ??**).

TABLE I
QUERY EXECUTION TIMES

TPC-H Scale	⏱ (sec)		
	ACTIVERECORD	**SWITCH**	SQL (✎)
0.001	0.814	**0.053**	0.007
0.01	8.112	**0.176**	0.044
0.1	81.290	**0.893**	0.595
1	DNF	**11.910**	12.162
10	DNF	**128.016**	119.794

DNF: overwhelms SQL parse buffer

Query execution times against *Spree* data derived from TPC-H instances of growing size. (Linux 2.6, 2.93 GHz Intel Xeon™, 70 GB RAM, SCSI disk, running IBM DB2 v9.)

for block invocation (`&:user_id` ≡ `{|x| x.user_id}`), remain available. SWITCH also embraces user-defined methods written in the same array-centric style: the singleton method `in_order()`, invoked in line 8 and defined as

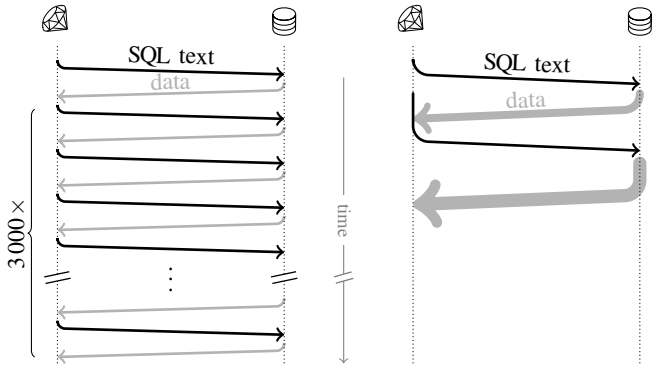
```
class << line_item = Line_Item
  def in_order(o) select {|li| o.id == li.order_id} end
end
```

encapsulates the *1-to-n* association between the *Spree* tables Orders and Line_Items, for example.

Abstract Interpretation. SWITCH discovers the structure of the source program through a variant of abstract interpretation [4] that we tailored to fit the RUBY execution model. SWITCH remains fully portable and does not intrude the RUBY interpreter or parser (as AMBITION [5] does, for example). In a nutshell, blocks `{|x| e(x)}` are invoked with crafted parameter objects that form an expression tree of all operations applied to them while the RUBY runtime evaluates expression *e* and its subexpressions. The actual value of *e* is only computed once SQL code generation and execution has been performed. In this respect, SWITCH bears resemblance with LINQ [9].

B. SQL Code Generation

Parts of SWITCH's compiler back-end for RUBY's array operations loosely follow the translation principles introduced by FERRY [7]. SWITCH relies on a new code generator designed to emit compact and readable SQL text. Code generation exploits SQL:1999 features like common table expressions (`WITH`), lazily forms full-select blocks only when the lack of compositionality in SQL:1999 forces it to, and avoids to sequentially ship the SQL text in a piece-by-piece fashion as we have observed with ACTIVERECORD in Figure 2b



(a) ACTIVERECORD. (b) SWITCH.

Fig. 5. Context switches and shipment of queries/result data between the Ruby runtime and the database back-end (against a TPC-H instance of scale factor 0.1).

(also see Section II-C). SWITCH turns the RUBY program of Figure 3a into the SQL statement of Figure 3b.

Query Performance. Deep query embedding and SQL code generation can still lead to competitive query performance. SWITCH–generated code can often contend with manually written SQL as we will demonstrate using a diversity of query classes and use cases taken from *Spree* and elsewhere (Section III). To make this point here, **?? ??** reports on an experimental comparison of SWITCH's SQL output with

(1) the SQL statement sequence generated by ACTIVERECORD 3.0.5 as well as
(2) a handcrafted query variant—see Figure 4 and column SQL (✎) in **?? ??**.

All measurements include the time required to materialize the query results on the RUBY heap.

The experiment indicates a performance gap of two orders of magnitude if ACTIVERECORD is compared to SWITCH or handcrafted SQL. Beyond this (admittedly shallow) "*faster is better*" assessment, for *Spree* application data derived from a TPC-H instance of scale factor of 1 or larger, ACTIVERECORD overflows IBM DB2 statement heap with SQL text of more than 10^7 characters (due to huge `IN (···)` clauses, see Section I) and thus fails to scale.

C. Busy But Bored Stiff — Shipping Fewer Queries

With ACTIVERECORD, the size of the database instance can affect the *size* of the generated SQL text. Worse, though, the database size may also determine the *number* of SQL queries generated. ACTIVERECORD partially addresses this phenomenon, also known as the *1+n query problem*, but RAILS still suffers [1, see `includes()`].

Consider the following RUBY piece that uses ACTIVERECORD to build a nested array of all orders together with their contained line items:

```
orders = Order.includes(:line_items)
orders.map {|o| [ o.id, o.line_items.order("price") ] }
```

Although we used `includes()` to materialize the association [1], ACTIVERECORD still issues *a separate SQL query per order*. A flood of simple look-alike queries[2] keeps the back-end busy and the overall execution time is dominated by costly context switches between the RAILS and database processes which repeatedly exchange SQL text and (tiny) pieces of data (see the illustration in Figure 5a).

Execution of the equivalent SWITCH formulation

```
Order.map {|o|
    { order: o.id,
      items: line_item.in_order(o).sort_by(&:price) }}
```

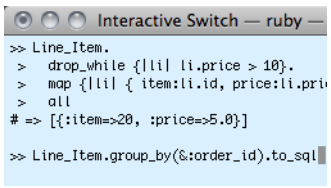
leads to a radically different interaction with the database back-end: independent of the database instance size, exactly two SQL queries will be executed. One of these queries retrieves *all* rows of table Line_Items in one go, tagging the rows such that an item's associated order object and the item's array offset are available to the RUBY process. The execution of these two queries may overlap arbitrarily (Figure 5b).

More generally, SWITCH guarantees to issue exactly n independent SQL queries to compute a result whose *type* features n `Array` classes. For the above example, the result type reads `Array⟨Integer,Array⟨Line_Item⟩⟩`, thus two queries are issued.

III. DEMONSTRATION SETUP

The software demonstration looks at SWITCH from the angles of a RAILS developer as well as a database programmer knowledgeable in SQL.

Trading ACTIVERECORD for SWITCH. SWITCH may be used as a drop-in replacement for the query functionality offered by ACTIVERECORD. We will demonstrate a live *Spree* Web shop instance in which code fragments have been adapted to use SWITCH. This will range from RAILS-typical code that populates HTML pages with database information to extensive pieces of RUBY code that have formerly not been considered database-executable (shopping cart checkout and payment). With SWITCH, new types of queries and functionality are easily expressed in RUBY and then added to *Spree*. We will demonstrate OLAP–style queries that perform credit card fraud analysis over *Spree* payment records.



Interactive SWITCH Shell. RUBY is an interpreted language that comes with an interactive shell (`irb`), enabling developers to experiment with snippets of code. SWITCH is as interactive as RUBY and we will provide an `irb` instance in which ad-hoc SWITCH-based programs may be formulated and executed. A specific `to_sql()` method allows the inspection of the SQL code that SWITCH generates. The (performance) advantage that comes with database-supported data-intensive programming—as opposed to ACTIVERECORD and traditional on-RUBY-heap array processing—is particularly easy to demonstrate in this interactive environment.

Try SWITCH. The demonstration will be hosted on a laptop computer that additionally acts as a local wireless LAN hotspot such that multiple users can try SWITCH independently. For any mobile web-enabled device (*e.g.* iOS-based or Android phones) this provides access to the *Spree* shop instance mentioned above. In addition, we will serve a web-based application that offers an `irb`-like interactive RUBY shell. Canned example programs are prepared but users may also formulate and evaluate their own queries. Optional pop-ups show the SQL code that is generated behind the scenes.

ACKNOWLEDGMENT

This research is supported by the German Research Council (DFG) under grant no. GR 2036/3-1.

REFERENCES

[1] ACTIVERECORD *Query Interface*. api.rubyonrails.org.
[2] R. Agrawal et al. The Claremont Report on Database Research. *CACM*, 52(6), 2009.
[3] P. Alvaro, N. Conway, J.M. Hellerstein, and W.R. Marczak. Consistency Analysis in BLOOM: a CALM and Collected Approach. In *Proc. CIDR*, 2011.
[4] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. POPL*, 1977.
[5] DATAMAPPER, SEQUEL, AMBITION. *Database Toolkits for* RUBY. rubygems.org.
[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.
[7] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-Safe LINQ Compilation. In *Proc. VLDB*, 2010.
[8] P. Hudak. Modular Domain Specific Languages and Tools. In *Proc. ICSR*, 1998.
[9] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Objects, Relations and XML in the .NET Framework. In *Proc. SIGMOD*, 2006.
[10] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proc. SIGMOD*, 2009.
[11] *Ruby on Rails: Web Development that Doesn't Hurt*. rubyonrails.org.
[12] *Spree: Open Source E-Commerce for Ruby on Rails*. spreecommerce.com.
[13] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proc. PLDI*, 2007.

[2] For every single order id $\langle o \rangle$, ACTIVERECORD issues the SQL statement `SELECT * FROM Line_Items WHERE order_id =`$\langle o \rangle$ `ORDER BY price`.