A Deep Embedding of Queries into Ruby

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät

der Eberhard Karls Universität Tübingen

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

(Dr. rer. nat.)

vorgelegt von

Dipl.-Inf. Manuel Mayr

aus Brixen (Italien)

Tübingen

2013

Tag der mündlichen Qualifikation:	05. Juli 2013
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Torsten Grust
2. Berichterstatter:	Prof. Dr. Marc H. Scholl

Eberhard Karls Universität Tübingen
Wilhelm-Schickard-Institut für Informatik
Mathematisch-Naturwissenschaftliche Fakultät

A Deep Embedding of Queries into Ruby

Dissertation
zur Erlangung des Grades eines

Doktors der Naturwissenschaften
(Dr. rer. nat)

vorgelegt von
Manuel Mayr

Betreuer
Prof. Dr. Torsten Grust

Tübingen, 2013

Zusammenfassung

Relationale Datenbanksysteme und Programmiersprachen sind von unterschiedlichen Paradigmen geprägt. Datenbanksysteme beschränken sich noch immer auf die Verarbeitung flacher Tabellen (ein Modell, das sich insbesondere für große Datenmengen bewährt hat). Dem gegenüber unterstützen moderne Programmiersprachen eine Vielzahl von Hilfsmittel und Abstraktionsmöglichkeiten, die den Entwicklern die Datenmodellierung erheblich erleichtern. Unter diesen Hilfsmitteln finden sich beispielsweise geordnete und verschachtelte Datenstrukturen samt entsprechenden Operationen, um diese zu verarbeiten.

Da Datenbanksysteme aus den meisten Webapplikationen nicht mehr wegzudenken sind, sind insbesondere Web-Entwickler dem ständigen Wechsel zwischen diesen beiden Paradigmen ausgesetzt. Die Kommunikation mit Datenbanksystemen macht es erforderlich, Anfragen in SQL zu formulieren, um auf persistente Informationen zurückzugreifen. Um die Informationen weiterzuverarbeiten, müssen diese in das Datenmodell der jeweiligen Gastsprache konvertiert werden. Es ist wenig überraschend, dass dieser Ansatz unweigerlich zu schwer wartbaren Programmen, sowie zu Performanzproblemen und sogar Sicherheitslücken führt.

Das Problem ist einfach: Es ist mehr als doppelt so schwierig zwei Programmiersprachen zu benutzen als eine.

(Cheney, Lindley und Wadler)

Mit den Techniken, welche wir in dieser Arbeit vorstellen werden, rückt die Lösung dieses Problems in greifbare Nähe. Wir wenden uns dem auf der Programmiersprache Ruby basierenden Web-Framework Ruby on Rails (auch Rails genannt) zu. Dabei werden wir eine Reihe von Problemen aufzeigen, welche auf ActiveRecord, die Datenbankanbindung in Rails, zurückzuführen sind. Unsere Aufmerksamkeit gilt vor allem der großen Ähnlichkeit zwischen ActiveRecord und SQL, eine Ähnlichkeit, die es mitunter unmöglich macht, Ruby Idiome in der Formulierung von Datenbankanfragen zu verwenden: (1) Anfragen werden im Wesentlichen basierend auf einem sehr einfachen Konzept formuliert. SQL Fragmente werden in Ruby als Zeichenketten formuliert und dann durch entsprechende Methodenaufrufe direkt in eine der SQL Klauseln eingefügt. (2) Weder geordnete noch verschachtelte Datenstrukturen und deren Verarbeitung werden *nicht unterstützt*. (3) Zudem werden in der Regel SQL Anfragen in Abhängigkeit der Daten erzeugt, welche angefragt werden. Dies führt mitunter zu erheblichen Performanzproblemen.

Mit Switch, einer *nahtlosen Integration einer Anfragesprache* in Ruby, verfolgen wir das Ziel, die Grenzen zwischen der Gastsprache und dem Datenbanksystem zu verwischen. Mit unserem Ansatz adressieren wir alle in

ActiveRecord auftretenden Probleme auf einen Streich: Mit Switch gibt es (1) *weder einen stilistischen noch einen syntaktischen Unterschied* zwischen Ruby Programmen, die auf Arrays im Hauptspeicher oder Tabellen in einer Datenbank operieren, (2) selbst wenn sich diese Programme auf Ordnung verlassen oder verschachtelte Datenstrukturen verwenden. (3) Der Übersetzungsmechanismus und der SQL Generator in Switch garantieren, dass nur wenige Anfragen erzeugt werden. Dies hilft uns dabei, die Performanzprobleme, welche mit ActiveRecord entstehen, zu mindern.

In der vorliegenden Arbeit werden wir jene Schritte erläutern, die es ermöglichen, Ruby Ausdrücke in semantisch äquivalente SQL Anfragen zu übersetzen. Wir werden eine Reihe von sorgfältig ausgewählten Ruby Konstrukten und Idiomen identifizieren, deren Semantik sich auch im Datenbankkontext modellieren lässt. Ein Mechanismus, der uns den Zugriff auf die Syntaxbäume dieser Konstrukte erlaubt, ebnet dabei den Weg, auf etablierte Übersetzungstechniken aufzubauen.

Aufbauend auf einer Übersetzungstechnik namens *Loop Lifting* werden aus den Ruby Ausdrücken Anfragepläne erzeugt. Diese Pläne bestehen vorwiegend aus Primitiven aus der klassischen Relationalen Algebra. Nur wenige Operatoren wurden hinzugefügt, um zum Beispiel geordnete Datenstrukturen korrekt im Kontext einer Datenbank abzubilden. Diese Vorgehensweise erlaubt es uns zudem auf eine bereits bestehende Infrastruktur zurückzugreifen, um die ungewöhnlichen Anfragepläne signifikant zu vereinfachen.

Mit einem SQL Generator beenden wir unsere Reise durch die verschiedenen Etappen der Übersetzung. Der Generator verwandelt die Anfragepläne in standard konforme SQL:1999 Anfragen, um von jenen Systemen zu profitieren, die für die Verarbeitung großer Datenmengen prädestiniert sind.

Es sieht aus wie Ruby*, ist aber so schnell wie handgeschriebenes SQL,*

ist das Ideal, welches die Entwicklung und Forschung rund um Switch antreibt.

Abstract

The mismatch between relational database systems and programming languages is a long-standing problem in the community. Whereas database systems still operate on flat tables, modern programming languages support a garden variety of features, including ordered and nested data structures, as well as data abstraction.

Particularly web developers suffer from this two-paradigms approach, because they must repeatedly undergo the mental shift from the host language to SQL (and vice versa) that enables them to communicate with the relational back-end. It does not come as a surprise that this approach inherently leads to unmaintainable code, performance issues, and even security holes such as SQL injection attacks.

> *The problem is simple: two programming languages are more than twice as difficult to use as one language.*
>
> (Cheney, Lindley, and Wadler)

With the techniques we developed in this work, the solution to this problem is close at hand. We will turn our focus on RUBY ON RAILS (RAILS, for short), a RUBY-based web-framework, and identify the key issues that trace back to RAILS's ACTIVERECORD database-binding. We will demonstrate that ACTIVERECORD's query interface is little more than SQL in disguise: (1) Query construction relies on a *simplistic, concatenative semantics,* (2) *order-* and *nesting-based* programming is *not supported,* and (3) the number and size of generated queries directly depends on the *queried data.*

Our efforts to provide a solution for the above problems assembled into SWITCH, a *deep-embedding* of queries into RUBY and RUBY ON RAILS, which aims to blur the traditional lines between the host language and the relational database back-end. Our approach tackles all of the above issues at a single stroke: With SWITCH, there are (1) *no syntactic or stylistic differences* between RUBY programs that operate over in-memory array objects or database-resident tables, (2) even if these programs rely on *array order* and *nesting.* (3) SWITCH's built-in compiler and SQL code generator guarantee to emit *few queries*, addressing performance issues that arise in ACTIVERECORD.

This thesis details the steps necessary to take a RUBY expression all the way down to SQL. We will identify a sensible set of native RUBY constructs, yet feasible to be translated into a datbase-executable form. A mechanism that enables us to turn RUBY expressions into a *runtime-accessible* expression tree paves way for a full-fledged compiler-backend.

To take the RUBY expression towards a database-executable format, we directly expand on a compilation technique, called *loop lifting.* The query plans resulting from this technique primarily consist of primitives that resemble the operators of the classical relational algebra, enriched with ranking

facilities to correctly reflect order. We take advantage of an existing optimization infrastructure designed to significantly simplify the unusual plan shapes stemming from the loop-lifted compilation.

With the SQL:1999 code generator we conclude our journey through the different stages of our compiler. The code generator turns intermediate algebraic plans into strictly standard-compliant SQL:1999 queries and lets them take advantage of one of the most capable systems for processing large-scale data available today.

Looks like RUBY, *but performs like handcrafted SQL,*

is the ideal that drove the research and development effort behind SWITCH.

To My Grandmother
Aloisia Antenhofer

It is my humble attempt at thanking her for
everything she has done for me in my life.

Acknowledgments

Several years have passed since I started to work at the Technische Universität München, when Torsten offered me the chance to become a part of his team. I gladly accepted and never looked back, although there were times when I was having difficulties believing in myself. During all this time, Torsten has never lost faith in me. His guidance and invaluable advice often saved me from getting stuck in a dead-end. I am deeply grateful for the constant support I have received throughout the last years. *Thank you so much, Torsten!*

I would like to thank all fellow contemporaries at the Technische Universität München and at the Eberhard Karls Unversität Tübingen for the thriving and lively setting they provided all through the last years. Particular thanks are due to Jan Rittinger, with whom I shared my office for a very long time. With his astuteness and his profound knowledge of relational concepts he could often help me to sort out my thoughts. Thanks for all the shared experience and the countless discussions.

Apart from the working environment, I owe thanks to numerous other people who have accompanied me along my path. Special thanks go to Andrea Reichhalter, Gregor Kovács, and Nicole Christiane Kaden for their company and friendship. Nicole also agreed to review this thesis and brought clarity to my writing. Further, I would like to thank my girlfriend Guliza for bearing with me for the last six months when I decided to spend more and more time at the university. All of you have become an important part of my life!

In fact, I have no words to express my gratitude towards my family. My mother and grandmother suffered most from my constant absence, but they never complained. I have no doubt that without their love and constant support this work would not have been possible.

I love you!

Contents

Chapter 1

Introduction

Ruby on Rails [Rub] (or Rails for short) has matured into one of the most popular web frameworks since it was released in the two thousand and four. Within the first months that followed its initial release, Rails achieved tremendous success that still continues. Nowadays it serves as the backbone for a variety of notable Web 2.0-style applications and services so rich in their complexity that they had been hardly realizable without the sensible features Rails offers. Among them are applications such as *Github*, *Qype*, *Twitter*, or *Xing* to name merely a few of those widely recognized on the Web.

Before David Heinemeier Hanson came up with Rails, web applications were homogeneous code blocks. Developers tended to intermingle routines that handle the client-facing presentation, application logic and the interaction with a relational back-end. As demands increased, they had to cope with the overwhelming complexity of legacy code that turned out to be unmanageable. This problem was the incentive for the development of Rails.

Ruby on Rails imposes some constraints on the design of applications. At a closer look, however, those restrictions prove to be a guideline to ease the development process. The architecture of every Rails-built application follows a strict *model–view–controller* [GH+94] (MVC) design pattern that enables developers to structure their applications in accordance with

Model	an interface with a relational back-end that serves application data and holds state information,
View	the client-facing presentation and interaction, typically based on HTML5/CSS/AJAX, and
Controller	the specification of the application's call interface, typically in terms of a REST-ful URL router.

These closely cooperating subsystems interact in RAILS as depicted in Figure 1.1: ① The interaction starts with a *request* from the client, typically a browser, sent to the web server. Following the reception of this request, the ② web server delegates it to the router, which identifies it and ③ dispatches it to a controller's action. The controller ④ queries and ⑤ gathers proper data from the model, and ⑥ supplies the view with the data in order to render a suitable representation for the client. ⑦ Then the view sends this representation back to the controller. ⑧ The controller hands the representation over to the web server, which in turn ⑨ sends it back to the client.

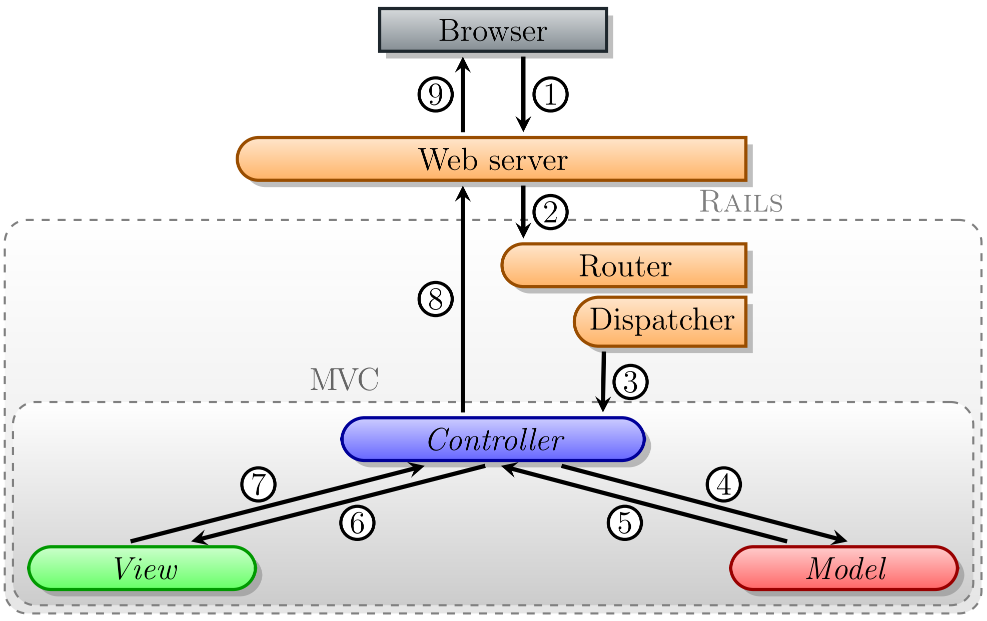


Figure 1.1: MVC as it is used in RAILS.

1.1 Programming Languages and Databases: Distant Shores

Even though relational database management systems (commonly abbreviated RDBMS) have evolved into efficient query processors over the last decades, the attempts to fully exploit these enhanced query capabilities were rare. In the Web community database systems are usually perceived as mere storage containers and operated as such. Most of the analytical capabilities are ignored. As we will see in the following sections, this does not come as a surprise.

Today's persistent systems are supported by disparate mechanisms based upon different concepts and paradigms. On the one hand, there is the programming language domain, which presents a Turing-complete environment, that encourages computation over arbitrary data defined using the language type system. On the other hand, there is the database domain, which from the programmer's angle is characterized by the following attributes:

Data Model. RDBMS have been designed with the mathematical concept of relations in mind. The rather simple n-ary relation, with the table as the conceptual representation, is the single aggregate structure in the relational model. Each element (or row) in a relation represents an n-tuple. Relations inherently lack the notion of order. Moreover, in the relational model, *positional addressing* has been replaced by *associative addressing* in a sense that each datum in a database can be uniquely identified in terms of its table name, attribute and primary value. Consequently, the placement of new data inserted into the database is completely left to the system.

In contrast to the relational data-model provided by database systems, programming languages intrinsically support a diversity of data structures, such as possibly nested or associative arrays enabling developers to shape data according to their requirements. The developer can now allow rely on several characteristics, such as order, that can significantly ease the manipulation and organization of data.

Query Model. On the basis of the relational data model, the relational approach calls for *set processing capabilities.* The primary purposes underlying the relational algebra is the avoidance of loops [Cod82]. Therefore, each of the relational algebra operators entails processing whole relations and thus transforms one relation into a new one. Much of the derivational capacity is obtained by the triade: project, select and join.

This compares to the *execution model* of a programming language. Found among all programming languages is some form *iteration* that directly reflects the capabilities of the underlying processing unit. A popular variant are the *enumeration-controlled loops*, which originated in the `do`-loop in Fortran [For]. A similar mechanism has been adopted by nearly every subsequent language. However, the syntax widely varies and ranges from *iterators* [GH+94, chap. 5, p. 257ff] in object-oriented languages to *functors* [Jon03, chap. 6, p. 87ff] in the functional paradigm.

SQL [MS01], the standard language of data retrieval and manipulation, largely adopts the set-processing capabilities of its formal underpinnings. The `SELECT`-construct in SQL for example embodies the project, select and join operators of the relational algebra in a single statement. As a *declarative language,* SQL leaves it to the system to take necessary steps to select proper access paths when retrieving or manipulating data.

1.1.1 Impedance Mismatch

The conceptual and technological differences between programming language and database domain are known as *impedance mismatch* [Mai90], which reflects the whole spectrum of these diverging systems. Cook and Ibrahim identified several problems that impede the integration of relational database systems into programming languages [CI05]: (1) imperative/object-oriented programs versus declarative queries, (2) algorithms versus tables and indexes, and (3) threads versus transactions. As a result, databases are both difficult to design and develop, and attaining good performance usually requires sensible optimization based on the knowledge of experts.

The disparity between programming language and database domain diverts developers' attention from the actual task by obliging them to focus on mastering the disparate paradigms. Hence, despite the frugality of the relational model application builders, *e.g.* RUBY developers employ intrinsic structures of their host language when formulating algorithms. In contrast to relations, arrays are ordered. Even arbitrary nesting can be induced into arrays and hashes. The mismatch between intrinsic data structures of the host language and relations is one to consider when interfacing a database system. Furthermore the aforementioned set-orientation of RDBMS is rather hard to grasp for a lot of developers. Developers tend to interface a database by scanning whole tables and materializing them on the host language heap, where they can rely on familiar concepts to additionally process the data. Not surprisingly, this is destined to fail because most host languages are not designed to cope with bulks as large as those that can be found in a database.

1.2 State-of-the-art Database Bindings

There have been various approaches to integrating queries into a host language environment. The degree of integration ranges from *orthogonal persistence* to *explicit query execution.* The latter keeps data formats completely disparate and there is no automated support for transformation between them. At this low degree of integration programmers need to cope with multiple representation semantics and the mappings between them. With orthogonal persistence, there is no visible distinction between data formats.

Typically the approaches are located in the middle of this range, where the application builder still has to specify mappings and understand the relationships between the multiple representations, but is relieved of the task of writing explicit translation codes. In the following two sections we present approaches on the margins of this range.

1.2.1 Call Level Interface

A widely known example for *explicit query execution* is JDBC [Jdb], which aims for database-independent connectivity between the host language Java and databases. JDBC was intended as a call level SQL interface for the Java platform by "focusing on the execution of *raw* SQL statements and retrieving their results [HC97]."

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
                     "SELECT id, name, salary"
                   + "  FROM Employees"
                   + " WHERE salary > 3000");

while (rs.next()) {
  int    id     = rs.getInt("id");
  String name   = rs.getString("name");
  float  salary = rs.getFloat("salary");
}
```

Figure 1.2: Table access via JDBC

In Figure 1.2, database access is explicit. The SQL query is embedded as a string in the program and executed on the backing store. The actual data is then gathered by iterating over the result (`ResultSet rs`), which provides an iterator interface to access the data collected by the SQL query. Attribute names are used to extract data from the result set.

This approach manifests a high degree of impedance mismatch, yet it is popular and in common use. Apart from JDBC, at the moment there exist several similar approaches, as *e.g.* Microsoft's ODBC [Ms] or Perl's DBI [Per], to name a few.

The primary mechanism for explicit query execution is the *call level interface* [VP95] (abbreviated as CLI). Characteristic for CLI is the execution of SQL queries through a standardized interface, in which queries are represented as strings in the program. However, call level interfaces present a number of significant problems we can identify in the above example: (1) The queries remain unchecked until runtime, because they are embedded as strings in the program. (2) Query results, represented as dynamically typed objects, again are accessed by strings.

1.2.2 Language-Integrated Queries

A better, yet in terms of orthogonal persistence, not fully satisfactory approach to integrating database technology into programming languages is to merge a full fledged query language and a host language like C#. Microsoft has pursued this goal with LinQ [BMT07; Lin], which is an extension to the .Net framework.

```
var result =
      Employees.Where( emp => emp.Salary >= 3000 ).
               Select( emp => new { Id     = emp.Id
                                  , Name   = emp.Name
                                  , Salary = emp.Salary } );
```

Figure 1.3: Accessing a table via LinQ in C#.

Although Figure 1.3 is semantically equivalent to Figure 1.2, the integration into to the host language C# has come a long way. The keyword `var` introduces a strongly typed variable, whose type is determined by the type inference of LinQ. The methods **`Where`** and **`Select`** in the above example are provided by LinQ and accept lambda expressions (x => expr), which allow the programmer to pass code rather than simple data to be executed. After having been type checked, the query is compiled to a database agnostic intermediate representation. Following the translation into SQL statements, the resulting queries are sent to the backing store to be executed. The result is then shipped back into the host language heap and materialized into a regular C# object that can be manipulated as such.

1.3 Contributions of this Thesis

This thesis focuses on the deep embedding of queries into the host language Ruby. Our approach makes it possible to evaluate substantial fragments of Ruby on database systems. The results of this work assemble into the layered system we named Switch, depicted in Figure 1.4:

As part of this ① we develop techniques to derive expression trees representing the source Ruby program at runtime. We will devise ② a type-sensitive rewrite system as well as ③ translation rules to compile these trees into an algebraic representation, based on *loop lifting* [Teu06; GR08; Sch08]. While this representation is independent of a specific database system, it is susceptible to a variety of

④ optimization techniques that may be applied to the algebraic plans (a detailed discussion of these techniques is beyond the scope of this theses and can be found in [Rit10]). In addition, we provide a ⑤ translation scheme for compiling the algebraic query plans into database executable SQL:1999 statements. The resulting tables are then ⑥ materialized in the Ruby heap in the form of arrays and hashes.

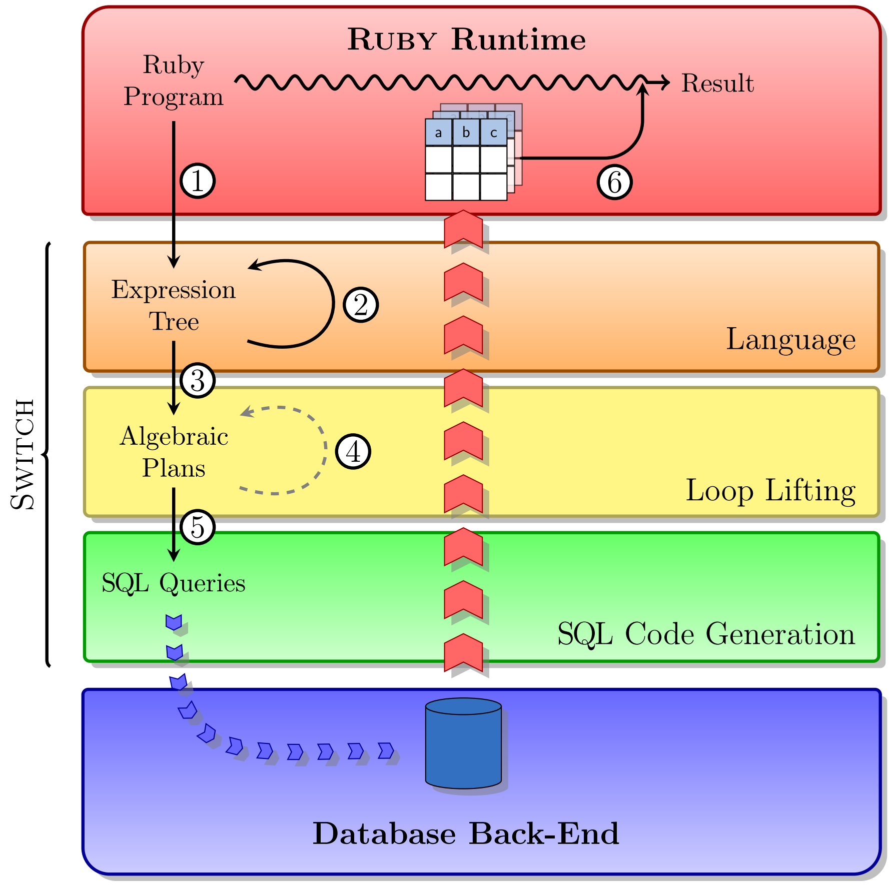


Figure 1.4: The above picture shows the layers that take part in the translation of a Ruby fragment into a database-executable SQL query. The blue and red signals (⌦) respectively denote query shipping and result shipping.

1.4 Outline

In the following, we will briefly discuss the overall structure and the topics we will cover in the course of this work:

Chapter 2: Query Integration into Ruby.
We start with an overview of the principal features of ACTIVERECORD, the state of the art database abstraction layer used in RAILS. In the course of this, we will discuss the translation strategy that ACTIVERECORD pursues to turn the RUBY expressions into a database-executable format.

We will then shift our focus to SWITCH, the query compiler that we devise in this work. Following the identification of RUBY fragments suitable to be executed on a relational database back-end, we proceed with a thorough description of the language features. With a proper type system, we will then lay the foundation for type sensitive rewrites, which enable us (i) to automatically coerce tuples into lists (depending on the context) and (ii) remove tuple specific functions at compile time.

Chapter 3: A Relational Portrayal of SWITCH.
In this chapter, we start off with a refresher on loop lifting. We then jump directly into the translation of SWITCH expressions. The expression are instantly translated into a variant of the classical relational algebra, which assumes the role of an intermediate representation. The choice for relational algebra as intermediate language has its particular strength. The relational operators are well defined and independent from a specific database back-end. At the same time, the algebraic primitives sufficiently reflect the query capabilities of modern RDBMS. Particularly, the mathematical characteristics of the relational primitives offer additional potential for a variety of optimizations.

In the following, we consider each SWITCH construct separately to provide a semantically equivalent relational expression. This translation faithfully deals with *tuples*, *records*, and (nested) *lists* and associated operations. Additionally, the semantics of order-sensitive operations is supported and preserved throughout the entire calculation.

Chapter 4: SQL Code Generation.
SQL as the standard query language represents an ideal starting point to target a variety of commercial database systems. In this chapter we describe the process of translating the algebraic plans (from Chapter 3) into SQL statements. We start with a simple translation scheme to illustrate the distinguishable features of these two paradigms.

In the following, we refine the translation. The result is a set of rules that partition each algebraic plan into a set of tiles that possibly contains several algebraic primitives. Each tile is then collapsed into a single SQL statement to jointly implement the original semantics of the algebraic plan.

Chapter 5: Assessment.
Here, we will see how SWITCH can benefit from the advanced optimizers participating in the query compilation of modern relational database systems. In the beginning of this chapter we will take a look at some of optimization strategies that can be applied to the algebraic plans derived from the loop-lifted compilation.

We then introduce a diversity of query classes, each of which is formulated in three variants: (i) an ACTIVERECORD formulation, (ii) a SWITCH formulation and (iii) an SQL formulation. We will dissect these queries and observe how they are executed on the database system. In the course of this, we will inspect and explain the factors that affect the database system to decide on a specific execution plan.

In the concluding experimentation section we will run the variants against each other and database instances of various size to measure the evaluation time. We will see that the SWITCH formulations can compete with the SQL formulations.

1.5 Prior Publications

Parts of this work have been published, or indirectly benefit from the following papers. I would like to thank Torsten Grust, Tom Schreiber, Jan Rittinger, Jens Teubner, Sherif Sakr, and Simone Bonetti for the fruitful collaboration.

T. Grust and M. Mayr. “A Deep Embedding of Queries into Ruby”. In: *Proceedings of the 28th IEEE International Conference on Data Engineering.* ICDE 2012. 2012, pp. 1257–1260

T. Schreiber, S. Bonetti, et al. “Thirteen New Players in the Team: A Ferry-based LINQ to SQL Provider”. In: *Proceedings of the 36th International Conference on Very Large Data Bases.* VLDB 2010. 2010, pp. 1549–1552

T. Grust, M. Mayr, and J. Rittinger. “Let SQL Drive the XQuery Workhorse”. In: *Proceedings of the 13th Int’l Conference on Extending Database Technology.* EDBT 2010. 2010, pp. 147–158

T. Grust, M. Mayr, et al. “Ferry: Database-Supported Program Execution”. In: *Proceedings of the 28th ACM SIGMOD Int’l Conference on Management of Data.* SIGMOD 2009. 2009, pp. 1063–1066

T. Grust, M. Mayr, and J. Rittinger. “XQuery Join Graph Isolation”. In: *Proceedings of the 25th Int’l Conference on Data Engineering.* ICDE 2009. 2009, pp. 1167–1170

M. Mayr. “Pathfinder meets DB2”. In: *Ph.D. Workshop of the 11th Int’l Conference on Extending Database Technology.* EDBT 2008. 2008, pp. 59–64

T. Grust, M. Mayr, et al. “A SQL:1999 Code Generator for the Pathfinder XQuery Compiler”. In: *Proceedings of the ACM SIGMOD Conference on Management of Data.* SIGMOD 2007. 2007, pp. 1162–1164

Chapter 2

Query Integration into Ruby

The conceptual backbone of each software system is a *domain model*. Domain models comprise the entities, attributes, and relationships which govern the problem domain. Since the advent of the object-oriented paradigm in the mid eighties, this programming model has been predominantly used to build complex applications. Object oriented languages provide typed objects consisting of state, behavior, and references to other objects and hence endow an ideal environment to design rich domain models.

Typically, objects from the domain model need to outlive the process that created it. This persistence aspect of applications demands for a long-term storage architecture, such as relational database management systems. Mapping objects to RDBMS is of critical importance to persistence applications. In contrast to object-oriented programming languages, RDBMS are equipped with tables of tuples and foreign key constraints to establish relationships to other tuples.

A common approach to overcoming the differences between object-oriented languages and RDBMS is *object-relational mapping* (or ORM for short). Fowler identified two widely used architectural patterns [Fow02, Ch. 10] that can be applied to persistent systems (Figure 2.1 depicts the basic difference):

(1) An ActiveRecord is a domain object in object-oriented programming that mimics a row in a database relation. Additionally each domain object is equipped with the ability to update, insert or delete itself from the database.

(2) DataMapper is a software layer that separates a domain object from its underlying database. Its role is to transfer data between these two storage layers. In contrast to ActiveRecord, the domain objects are not even aware of a persistent storage, such as a database system.

Rails heavily relies on ActiveRecord to interface a relational database system. The chapter starts with a description of the features and characteristics of

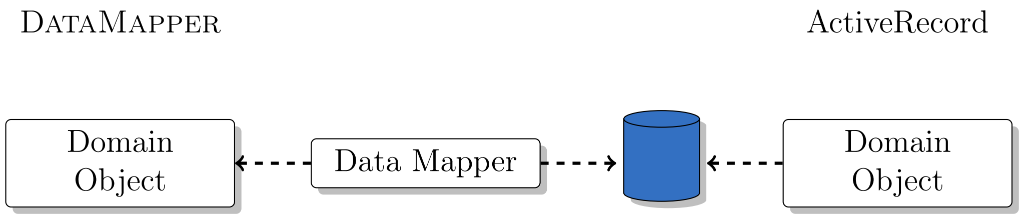


Figure 2.1: With DataMapper, the domain objects are completely separated from the persistence layer. Their purpose is to apply domain logic to the data while the interaction with the persistent store is completely left to the mapper objects. With ActiveRecord, the domain objects are furnished with the ability to communicate directly with the database system to gather relevant data. Additionally, they are equipped with behavior in form of *methods* in order to process this data accordingly (domain logic).

ActiveRecord. In particular, we will see how under the regime of ActiveRecord, queries are constructed through the chained invocation of *query methods*. We demonstrate that ActiveRecord is little more than SQL in disguise and conclude this critique of ActiveRecord with a demonstrating example of the mechanism that is used to turn these queries into SQL statements.

In the second part of this chapter, we present Switch, a deep embedding of relational queries into Ruby, and Ruby on Rails. Switch may be used as a drop-in replacement for the query functionality offered by ActiveRecord. We will identify a language fragment of Ruby that is suitable to be translated into relational queries. In particular, we will underline that with Switch there is no syntactic or stylistic difference between Ruby programs that operate over in-memory array objects or database-resident tables.

Following the description of the principal language components, we will turn our focus to a number of type-sensitive rewrites that help us to adopt the lenient syntactic conventions of Ruby. This leads us to an approach in which arrays can be regarded as either tuples or lists. On the database back-end, this classification determines the representation of an array either as a row (in case of a tuple) or as a column (in case of a list). To support this functionality, we introduced polymorphic functions into the language that can handle both tuples and lists along with a type system that can decide which variant is eventually used. Moreover, we provide a rewrite mechanism based on subtyping to automatically convert tuples into lists if appropriate.

2.1 A Survey of ActiveRecord

ActiveRecord is the primary persistence mechanism that is used in Rails. As an *object-relational mapper* ActiveRecord closely follows the standard ORM regime of a closely linked domain model and database schema: (1) Each domain class has a corresponding table in the database schema, (2) each object attribute is mapped to its corresponding columns and (3) instances of an ActiveRecord class resemble individual rows in a table.

In Rails a connection between a class and a table is established by inheriting its functionality from `ActiveRecord::Base`. Characteristic for Rails is the convention that a class named ⟨T⟩ (singular) is instantiated for any base table named ⟨T⟩*s* (plural). In the case depicted in Figure 2.2, a Rails application will refer to T, rather than T*s* (Rails comes with a set of rules for inflecting plurals for English words).

Another peculiarity, as shown in the figure below, is that class T does not mention any of the columns in table T*s*. ActiveRecord determines them dynamically at runtime, by reflecting on the schema inside the database. The corresponding *getter* and *setter* methods are then generated and bound to an instance of T without any further action by the developer. As a consequence it becomes possible to access fields, by standard Ruby methods t`.a` and t`.a = `a (to read and write column `a`) for any instance t of class T.

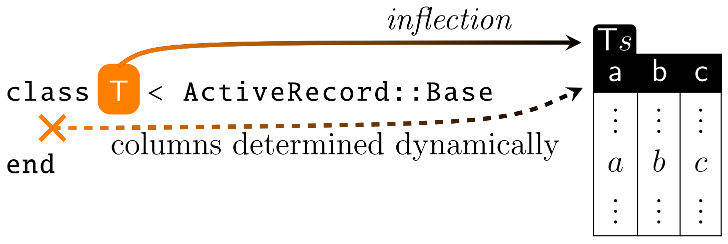


Figure 2.2: ActiveRecord maps a class name to a table name whereas column names are determined dynamically at runtime.

Whereas Ruby immanently uses dynamic typing—objects are defined by their call interface only—a column in a database table is associated with a type that determines the sort of the values contained in it. ActiveRecord gleans type information about those columns from the database schema and considers this information in order to map values taken from a column to instances of the corresponding classes in Ruby—*e.g.*, integer columns are mapped to instances of class `Integer` in Ruby. However, because Ruby is dynamically typed, when a row is brought into the Ruby heap, type information is neglected as long as the user does not update a field with an inappropriate value. When attempting to write the object back to the table ActiveRecord converts this value to the expected type unnoticeably, rather than preventing the user from performing an unsafe action. If the type of the value and column are incompatible, ActiveRecord falls back on default values, such as the number zero for numeric column types.

2.1.1 Dynamic Finder Methods

Whereas an ActiveRecord class embodies a table in a database, domain-model objects are akin to individual rows in a table. In addition to automatically generated getter and setter methods each class is equipped with *dynamic finder methods*. Finder methods are the simplest query abstraction that can be found in Rails. Their purpose is to retrieve rows from the back-end and convert them to a domain-model object. Generally, such methods wrap commonly used SQL queries and return ActiveRecord objects.

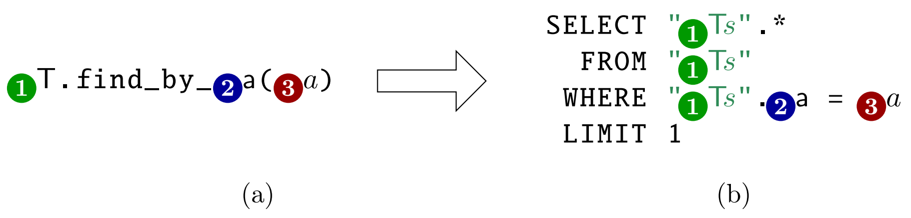
A finder method, such as `T.find_by_a(`a`)` in Figure 2.3a, delivers exactly one object required to meet the criterion that the value in field `a` must match the parameter a. Figure 2.3b shows the SQL statement that ActiveRecord will derive when invoking a finder method in Rails.

Essentially, the invocation of a finder method breaks down into *two phases*. In the following description, we refer to the finder method `T.find_by_`$\langle$`a`$\rangle$`(`a`)`:

Collection Phase. All decisive parts of the finder method must be collected: ❶ The plural form of the model name `T` (namely `T`s) serves as the name for the base table (`FROM`) in the database back-end. ❷ The postfix $\langle$`a`$\rangle$ of the finder method, which is used to determine the column to be checked, ❸ together with the value a are turned into a filter condition (`WHERE`) and woven into the template SQL query shown in Figure 2.3b. Only rows whose column `a` matches the value a are allowed pass the condition and finally appear in the final result. Additionally, the `LIMIT` clause confines the result to exactly one row.

Retrieval Phase. After the placeholders denoted by ❶, ❷ and ❸ in the template SQL query on the next page have been substituted, the query is sent to the database system. The resulting data is then turned into an array of Ruby hashes `[ { a:`a`, b:`b`, c:`c` }, … ]` by the database driver and presented to the call level interface of ActiveRecord. In conclusion, because we expect a single row, the hash is extracted from the array and converted into an instance of the class `T`.

Finder methods are the most elementary query abstractions that can be found in ActiveRecord. However, the techniques described here already convey an idea of how ActiveRecord operates when constructing SQL queries.



```
①T.find_by_②a(③a)
```

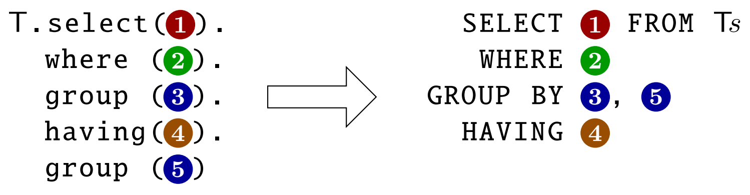
```
SELECT "①Ts".*
  FROM "①Ts"
 WHERE "①Ts".②a = ③a
 LIMIT 1
```

Figure 2.3: Placeholders in the SQL query denoted by ①, ② and ③ are substituted by the corresponding parts of the finder method on the left.

2.1.2 Query Methods

The query capabilities of ActiveRecord have come a long way. In past versions, developers fell back on variants of finder methods to express more complex queries. In version 3.0 and onwards `ActiveRecord::Relation` has emerged as the primary query abstraction. In the following, we refer to instances of the class `ActiveRecord::Relation` as *query objects*.

Under the regime of `ActiveRecord::Relation`, queries are constructed by the chained invocation of specific *query methods*. The method invocation chains originate in table classes T reflecting the base tables present in the underlying database back-end; in fact, the table class T delegates query methods to a blank query object.



```
T.select(①).
  where (②).
  group (③).
  having(④).
  group (⑤)
```

```
   SELECT ① FROM Ts
    WHERE ②
 GROUP BY ③, ⑤
   HAVING ④
```

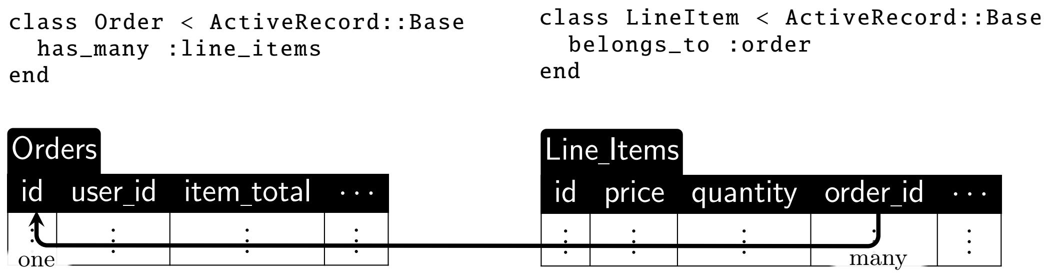
Figure 2.4: SQL query clauses are gathered by the chained invocation of query methods.

Each query object simply resembles an SQL query. Query methods—such as `select`, `group`, `where`—implement a concatenative semantics of query construction, *i.e.*, an `ActiveRecord::Relation` object gathers query clauses of a `SELECT`-statement as shown in Figure 2.4; query objects maintain a list for each SQL clause. When invoking a query method o.q(x), the calling query object o is cloned along with its SQL clauses. Depending on the query method q, the formal argument x is tracked in one of the lists of the cloned query object.

As with finder methods in Section 2.1.1, all the decisive parts of the SQL query are gathered before the query can be sent to the database system. Whereas in finder methods the data retrieval phase takes over directly after invocation, query objects collect SQL clauses successively by chaining query methods. Only when the result is going to be consumed—for example through the invocation of a method that enumerates the result, such as `each()` or `map()`—an attempt to construct executable SQL text from the clauses gathered so far is triggered.

2.1.3 Relationships

Most applications are based on domain models with entities that have a relationship to other entities—for example, an order includes at least one line item. Within a database schema these relationships are expressed by *foreign-key relationships.*



```
class Order < ActiveRecord::Base
  has_many :line_items
end
```

```
class LineItem < ActiveRecord::Base
  belongs_to :order
end
```

Figure 2.5: Associations in ACTIVERECORD

Likewise, ACTIVERECORD features the notion of associations[1] between its domain-model objects, and more important, the knowledge about these relationships can also be employed in queries. Relationships are established in a class by declaring them via `has_one`, `has_many` and `has_and_belongs_to_many`, which specifies one-to-one, one-to-many and many-to-many relationships respectively. Another specifier, `belongs_to`, is used in a one-to-one or one-to-many relationship to denote membership to another object; in other words, the model for the table that contains the foreign key always includes the `belongs_to` declaration. Figure 2.5 illustrates how relationships are modeled in both a database schema and ACTIVERECORD. An example of how associations are used in a query and how they translate into SQL queries shall be given in Section 2.2.

[1]In object-oriented parlance, an association between objects is similar to a relationship between tables.

2.2 A Critique of ActiveRecord

Even though ActiveRecord comes with a variety of query methods, instances of `ActiveRecord::Relation` are little more than an SQL in disguise: SQL clauses are gathered by chaining query methods; depending on these clauses, an executable query is constructed as soon as the result is going to be consumed.

The following discussion will revolve around *Spree*—a versatile Rails framework to construct E-Commerce applications [Spr]—to shed some light on how queries are formulated using ActiveRecord in a concrete setup.

2.2.1 Spree: A Rails-Based E-Commerce Platform

A *Spree*-generated web shop supplies operators with core functionalities—such as maintaining a collection of products and customers, and keeping track of the orders made by the latter. *Spree* is set up on a new Rails application that is enriched by a flexible domain model and customizable views, designed to run a web shop "out of the box." The domain-model data reside in relational tables, whose layout closely resembles the TPC-H benchmark (Figure 2.6 shows an excerpt).

Orders

id	user_id	item_total	···
⋮	⋮	⋮	⋮

Users

id	name	···
⋮	⋮	⋮

Line_Items

id	price	quantity	order_id	···
⋮	⋮	⋮	⋮	⋮

Figure 2.6: Tables containing *Spree* domain-model data (excerpt). The id columns serve as primary keys, order_id and user_id references the Orders and Users table respectively.

> *What would be the cost of granting a 10 percent discount to the open orders placed by all high-volume customers?*

As a Web-Shop operator this might be a pressing strategic question that must be answered. The Ruby snippet in Figure 2.7 answers this question using the ActiveRecord approach to query embedding. The formulation is fourfold:

(1) We define a high-volume customer to be a user who hash placed more than ten orders in the webshop; the variable **`high_vol`** (line 2) reflects this. High-volume customers are determined in lines 5 through 8. The orders are

```
 1  discount = 10.0/100 # grant a 10 percent discount ...
 2  high_vol = 10       # ... to customers with more than
 3                      # 10 open orders
 4
 5  high_vols =     Order.group("user_id").
 6                        having(["COUNT(user_id) >= ?",              ①
 7                                high_vol]).
 8                        select("user_id")
 9
10  open_orders =   Order.where(["user_id IN (:tc) AND state = :s",
11                              { tc: high_vols.map(&:user_id), ④    ②
12                                s:   "O" }])
13
14  items =         open_orders.includes(:Line_Item).                 ③
15                              map(&:line_items).flatten
16
17  cost =          items.sum {|i| i.price * i.quantity} * discount
```

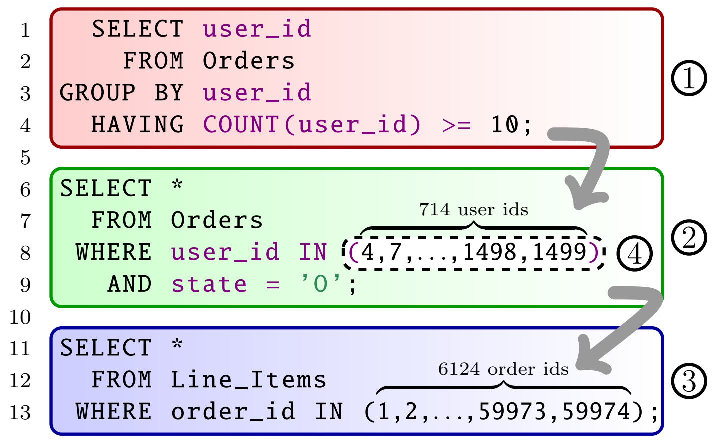
Figure 2.7: RUBY code written in ACTIVERECORD –style. The red code fragments are not considered to be database-executable.

grouped according to the `user_id` that uniquely identifies a customer; hence, each group contains the orders made by a customer. The criterion applied to each group in line 7 ensures that only groups containing more or equal `high_vol` orders can pass our filter.

(2) Following the identification of proper customers, their outstanding orders (state is equal to `"O"`) must be found. This is achieved in the `where()` method in line 10. Note that the result of the preceding query is used to take only high-volume customers into account. Methods, such as `map()`, consume the query result and thus trigger an attempt to materialize the data from the back-end into the programming language heap; that is where `map()` is then executed.

(3) Next, in line 14 all line items belonging to the orders are gathered using an association via the query method `include()`. Again, the line items are obtained by iterating over the open orders via `map()`. Because each order comprises several line items, this leads to a nested list which is then flattened (`flatten()`) to simplify the following calculation.

(4) Finally, the aggregated sum of price and quantity of the line items is used to calculate the estimated cost of a 10 percent discount.



```
  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 = '0';

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

Figure 2.8: SQL statement sequence generated for the Ruby snippet of Figure 2.7. Violet code is copied verbatim.

2.2.2 Style Matters!

Even though the Ruby snippet shown in Figure 2.7 is workable and delivers a correct result, it does not bear many similarities to a well-written Ruby program. The code is sprinkled with SQL text fragments (in quotes "...") and displays resemblance with SQL queries.

In the argument of the `where()` method in line 10 parameter markers (`?`, `:tc`, `:s`) are used to weave Ruby values into the query clause; query construction those markers are substituted by the corresponding values in the Ruby hash. Although the values are sanitized before sending them to the database, the resulting queries are not sufficiently[2] shielded against SQL injection attacks [WS07].

Overall, the query capabilities of ActiveRecord gives the impression that SQL mastery is indispensable to write efficient Ruby programs interacting with a database back-end. Query methods are convenient to insulate developers from using SQL in their daily routine—when searching for particular objects in a table (one object = one row) using simple filters—but as the queries become more complex they are unable to further assist you.

2.2.3 SQL Translation

The rails metaphor that Ruby on Rails adopted for its programming style based on conventions also applies to query construction: "query construction on rails." Like Rails in general, query construction is also pervaded by conventions, to

[2]While this thesis is being written a SQL injection vulnerability regarding the current Ruby on Rails versions (http://seclists.org/oss-sec/2012/q2/504) was reported.

that developers are encouraged to stick to. Though adhering to certain principles may lead to remarkable productivity boosts when rushing through the every day routine, it becomes difficult to move "off rails" to overcome the restrictions enforced by conventions.

All things considered, the aforementioned cases result in programming patterns like those in Figure 2.7. There, Ruby methods operating on native data structures of the programming language (`map()`, `flatten()`, `sum()`) are intermingled with ActiveRecord queries. 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 back-end. Interspersing native Ruby methods with ActiveRecord queries entails an execution context switch from programming language to the database query-processor and back. This phenomenon can be found in Ruby's execution traces.

The queries embedded into the Ruby source code via ActiveRecord in Figure 2.7 (denoted by ①, ② and ③, respectively) lead to the corresponding SQL queries in Figure 2.8. Note that the violet code fragments are literally adopted in the SQL clauses. The queries are merely used to bring a set of objects into the Ruby heap and leave the time consuming tasks—identified by the red code fragments—to the programming language, which, in most cases, is overwhelmed by the sheer volume of data it is facing.

Another aspect that affects the SQL queries on the preceding page is that the Ruby heap is used to carry intermediates of considerable size from query to query (see the arrows ↝). The resulting huge `IN (···)` clauses may easily overwhelm the back-end's SQL parser. The user ids identified by the values list ④ are generated by the equally identified fragment in the Ruby source code that is the result of query ①; note that `map` triggers ActiveRecord to construct the corresponding query. In contrast to the user ids, the `IN` clause containing the order ids is caused by the fragment ③ in the Ruby source, using `include(:Line_Item)` to determine the line items to each order. Based on the open orders, the identifier list is derived as the result the of query in ②.

2.3 Natural Query Embedding for Ruby

Ruby is an object-oriented script language with dynamism as part of its philosophy; in Ruby virtually everything may be altered at runtime:

Dynamic Typing

In Ruby, objects as well as classes in Ruby are defined by their *call interface* exclusively, *i.e.* the methods they are prepared to respond to when called on. When, due to a method invocation $o.m(x)$, a message is sent to an object the method m is searched *at runtime* within its class definition. If the class lacks a method m with a single argument x Ruby continues the lookup in the next class in the ancestor chain until the method is found. This behavior is known as *late binding*: the decision to retrieve a method is made at the latest possible moment.

Recall that objects in Ruby, despite being instances of a certain class, are not subject to type checking based on classes; as long as an object responds to the method being called upon, Ruby will not complain. In Ruby this is called *duck typing*.

Open Object Model

Another facet of Ruby is that any object may be extended by a new functionality at runtime[3]. Note that classes in Ruby are objects as well. Internally it does not distinguish between objects and classes; classes are mere objects with method containers that act as object factories.

The following mechanism is involved in which an existing method is altered or a new method is added to an object at runtime: Let o be an instance of class C. Both altering an existing and adding a new method cause a new class C' to be introduced and inserted between o and its original class C. The object o is now an instance of class C', and C becomes the immediate ancestor of C'. Hence, when invoking method m' on o, the lookup for m' begins in C' rather than in C. Ruby keeps these details to itself rather than revealing them to the programmers, who are not even aware of the existence of C' [4].

Metaprogramming

Altering the behavior of a program within the program itself is called metaprogramming. Lisp was the first language to employ this technique. Lisp is homoiconic, *i.e.* the primary representation of a program is accessible as a primitive data structure in the program itself. In Lisp both data and a program are represented as simple lists and thus easily modifiable by constructs of the language itself.

Ruby's design is heavily influenced by Lisp[5], so it is no surprise, they share the same virtues. Even though Ruby is not considered to be homoiconic, it is almost as malleable as Lisp. Ruby exposes some of the internals of its runtime model in the form of *hooks* that may be harnessed to react to certain events, such as the invocation of unsupported methods on objects. Furthermore, Ruby's open object model enables any program to adapt itself

[3]Rubyists often refer to this as "monkey patching".
[4]Because of its secret nature C' is called *eigenclass*.
[5]http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/179642

to its environment to build flexible APIs. For example, the design of ACTIVE-RECORD is heavily influenced by the metaprogramming abilities of RUBY: the tight coupling of ACTIVERECORD classes with the database tables prompts the classes to automatically define corresponding field names for each column.

Another valuable language feature that is coupled with metaprogramming is RUBY's ability to analyze itself; in programming languages this is called *reflection*. Reflection enables a programming language to examine itself during runtime. Programs can discover the following informations about themselves: (1) currently active objects, (2) class hierarchies of objects, (3) the attributes and methods of an object, and (4) information on methods.

RUBY's dynamic nature is closely tied to the idea of internal domain-specific languages, or DSLs [Hud98]. Internal DSLs are task-specific extensions of a general-purpose host language enabling the programmer to concisely specify solutions to a specific problem domain, and hence are only suitable only for a limited purpose. Usually such languages are equipped with primitives that are closely tied to the domain they are operating on, and thus make it easy for domain experts to both understand and formulate code. Despite having a long tradition in the computing landscape (particularly in LISP), DSLs have been neglected most of the time and have only recently gained attention with the advent of RUBY ON RAILS, which can be regarded as a set of domain-specific languages devoted to the seamless construction of web applications.

2.3.1 Collections and Enumerations

Since most programs deal with collections of data, RUBY is equipped with two built-in classes: arrays and hashes. Because these structures are so pervasive in most programs, they come with concise notations to construct literals.

Arrays contain a set of references to other objects. Each reference occupies a position in the array and may be indexed with a non-negative integer value, starting with `0`. RUBY is conveniently equipped with a concise notation to define literal arrays using square brackets. The following code snippet shows an example of how arrays may be created and accessed (we use $\rightsquigarrow$ to indicate the value of the expression):

```
a = [1,2,3]; a[1] # ⇝ 2
```

Likewise, **hashes** are the RUBY-way to provide *associative arrays* (or *dictionaries*). They are similar to arrays; the difference between them is that a hash is not limited to non-negative numbers. It can be indexed with any type of object that bears some notion of equivalence. Hashes can be constructed using braces (`{}`) as the following snippet demonstrates, and, like arrays, their elements may be accessed with square brackets:

```
h = {:a=>1, :b=>2, :c=>3}; h[:b] # ⇝ 2
```

Enumerable

Because the design and elegance of the `Enumerable` module are influenced by two language features of Ruby, we shall briefly explain those in the next two paragraphs before we proceed to discuss it in detail. The features in question are *blocks/lambdas* and *mixin modules.* Both constructs are reminiscent of functional languages and lead to a natural programming style.

Mixin modules (or mixins for short) are one of the defining characteristics of Ruby and to some extent similar to type classes in the Haskell programming language, which is the foundation for polymorphism in the language. With mixin modules it becomes possible to share similar functionalities throughout otherwise unrelated classes.

Blocks and lambdas are a language feature inspired by functional programming languages. Although they have slightly different semantics, both may be seen as function objects encapsulating code that may be injected into methods in order to influence their behavior. Both blocks and lambdas exhibit closure semantics, *i.e.* they retain variables that are in scope when they are constructed, and make them accessible in their context. An example of block notation can be found in Figure 2.9.

The `Enumerable` mixin is a generic iteration facility applicable to any collection data-type that exhibits the properties of a monoid. The mixin furnishes collections with all well-known list homomorphisms [MFP91], such as methods for traversal, searching and sorting, which involve an iterative process. Actually, `Enumerable` reveals an iterative core in form of the method `each()` that successively visits all items in a collection, such as `map()`, `select()`, `take()`, `drop()`, to name a few.

When invoked with a block, `each()` successively unveils the items by passing them to the block and binding them to the block parameter. This is commonly used to bring all items to an enclosing scope where they can be further processed. Every method in the `Enumerable` module is built around this iterative core construct, as *e.g.* the method e`.map {|x|` e_{body}`}`, which iterates over the elements of collection e_1 and applies the annotated block to every item in order to form the resulting array. The code snippet in Figure 2.9a demonstrates how `each()` is used. The

```
module Enumerable
  ...
  def map(&b)
    a = []
    self.each { |x| a << b[x] }
    a
  end
  ...
end
```

block notation

(a) Implementation of `map` in the `Enumerable` module.

$$\texttt{[1,2,3].map }\underbrace{\texttt{\{|x| x*2\}}}_{f}$$

$$\overset{*}{\rightsquigarrow} \texttt{[}f\texttt{(1),}f\texttt{(2),}f\texttt{(3)]}$$

$$\overset{**}{\rightsquigarrow} \texttt{[2,4,6]}$$

(b) Sample call of `map` on an array ($\overset{*}{\rightsquigarrow}$ denotes the intermediate, $\overset{**}{\rightsquigarrow}$ the final result)

Figure 2.9: Implementation and sample call of `map` on a collection.

items of the collection, denoted by `self`, are successively bound to the formal parameter `x` in the block followed by `each()`. The block `b` is applied to every item and its result then appended to the array `a`, the overall result of `map`. Note that `self` is bound to instances of the class that hosts the `Enumerable` mixin.

A characteristic shared by `map()` in the figure above and the other methods in `Enumerable` is that the items of any collection exposed by `each()` are inserted into an array on which the method-specific operation is performed. The same array is then returned as the result, so that multiple methods may be chained to a single expression—this is known as *method chaining* [FR10, Ch. 35]. This technique permits to compose rather complex algorithms using a concise notation. Due to anonymous functions in form of block notation in RUBY, this technique may easily lead to nested queries, such as the example depicted in Figure 2.10.

2.3.2 The Route from RUBY to Queries

In this work we present SWITCH, a natural query embedding that nestles the syntactic and semantic conventions of RUBY. Due to its dynamic nature, the language has been identified as an ideal host for internal domain-specific languages. Furthermore, RUBY already offers an iterative facility to perform complex operations on collections of data with `Enumerable` that adopts the concise syntax of list comprehensions [TP89]. Hence, we believe that a domain-specific language based on the `Enumerable` module querying against database-resident tables instead of collections in the programming language heap exhibits a perfect match between RUBY and database technology.

As already mentioned in Section 2.3, a DSL is a host language extension focusing on a particular domain. Usually a semantics model [FR10, Ch. 11] that captures the behavior of a system is covered by thin layers of such languages. The role of a DSL is to populate this model via a "parsing step." Note that the semantic model is an ordinary object model in the host language.

In Switch, the semantic model is understood as the abstract syntax tree (AST for short) that renders the structure of our input language. Whereas the front-end of a compiler uses lexical and syntactical analysis on a stream of text to arrange an AST, we rely on a series of method calls to assemble an equal result. The object model we use to represent the parse tree adheres to the interpreter design-pattern [GH+94, Ch. 5, p. 243ff]. The AST at hand paves the way for a full fledged compiler back-end concerned with static analysis, optimization and SQL-code generation.

Introducing an abstraction in form of an AST is a distinguishable characteristic of Switch, as opposed to ActiveRecord, in which a direct approach is employed whereby a SQL-query template has been chosen as the semantic model. This template is then populated by method chaining as shown in Section 2.1.

The Switch front-end is implemented as a module presenting an interface that resembles the one of `Enumerable`. However, in contrast to `Enumerable`, which performs operations on a collection, Switch is concerned with capturing the language structure in order to assemble the expression tree via the method call stack. Similar to the process depicted in Figure 2.9a on the facing page, we return a crafted object to support method chaining.

With an equally crafted object we invoke blocks (`{|x|` e(`x`)`}`) to form the expression tree of all operations applied to them while the runtime of Ruby evaluates e and its subexpressions. Using this technique, Switch does not intrude the host language interpreter as this is the case in other approaches that work directly on the parse tree of Ruby[6], such as Ambition [Amb]. Figure 2.10 shows an example of a Switch expression and its corresponding parse tree. Note that already in this phase any information that recalls the object-oriented nature of its host language has disappeared. The structure of the AST rather denotes the call to an ordinary function `map()`, which is supplied with two arguments (a nested array and an abstraction), than a method call. The same applies to the AST node that denotes the call to the function `max_by()`. We will describe the rationale behind this in Section 2.5.

In Ruby each identifier with a capital first letter is a constant. Classes and modules are no exception. In Switch, referring to a table R in the database back-

[6]In Ruby 1.8 it is possible to extract the parse tree of a method (`http://blog.zenspider.com/blog/2009/04/parsetree-eol.html`). This possibility has been dissipated with the advent of Ruby 1.9 due to internal changes in the interpreter.

```
𝒮([[1,2],[3,4,5]]).map {|x|
  x.max_by {|t| t}
} # ⇝ [2,5]
```

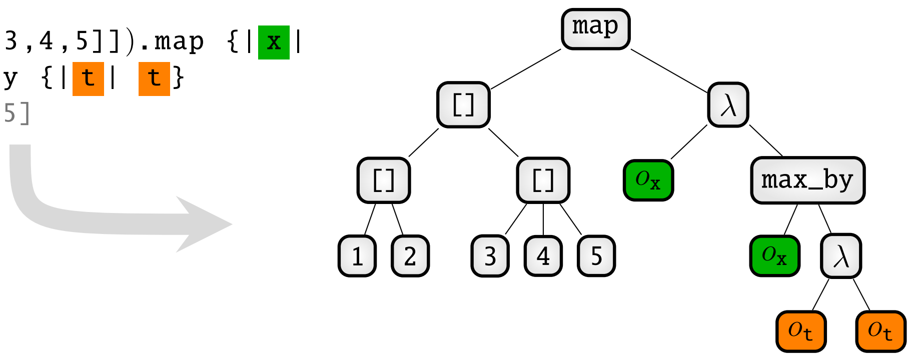


Figure 2.10: Arranging the parse tree implicitly over the method call stack. The green and orange highlighted fragments mark the block variables and the corresponding objects we introduce in SWITCH to capture the expressions applied to them.

end is accomplished via constants. We leverage the possibility to customize this constant-name–resolution algorithm, so that SWITCH gathers the database metadata and obtains the schema for the table R. If available, an object that represents the underlying table is generated and bound to the constant R. This object comprises the following information: (1) table attributes and the types associated with them and (2) information about constraints, such as primary keys. Commonly such table objects introduce a query in RUBY because it entails all methods applied to them executed under surveillance of SWITCH's capturing mechanism.

However, if a SWITCH query starts with a literal, such as in Figure 2.10, this literal needs to be wrapped by a function ($\mathcal{S}(\cdot)$) in order to turn it into a query object that is under SWITCH's surveillance. For the sake of readability, we omit this function in the following examples.

2.4 Idiomatic Ruby

RUBY supports several *programming idioms*, which are often encountered in programs. Much like in natural languages, idioms are an important part of any programming language and RUBY does not make an exception here [Ful06]. In fact, idioms are one of the reasons why RUBY is generally perceived as a very elegant and concise language.

The way in which we turn a RUBY expression into a runtime-available expression tree does not interfere with using idiomatic RUBY. Developers may thus continue to use idiomatic constructs in the formulation of SWITCH queries. Most of these constructs are not even recognized by our compiler because they are instantly translated into core constructs that the compiler already knowns how to deal with. In the following we will give you an overview of RUBY specific idioms that remain available in the formulation of queries:

Lambdas. We have already mentioned in the preceding section that Ruby supports anonymous functions, called *lambdas*. The notation

```
inc = ->(x) { x + 1 }
```

defines such a function with a single parameter and assigns it to the variable `inc`. The construct `&` enables a user to seamlessly convert these function into a block, so that the following expressions are equivalent:

e`.map(&inc)` $\equiv$ e`.map {|x| x + 1 }`

In this form, developers may use data abstraction as part of their queries.

Symbols. Ruby 1.9 adds the method `to_proc()` to the `Symbol` class. Similar to the dealing with lambdas described above, this method allows a symbol to be prefixed with `&` and passed as a block to a function. The symbol is then assumed to be the name of a method, and neatly turned into a block:

e`.map(&:`m`)` $\equiv$ e`.map {|x| x.`m` }`

Pattern Matching Ruby allows for a simple form of pattern matching in the block notation. Consider the following simple expression enabling the user to access the elements of the inner array in a very natural way:

```
[[1,2],[3,4]].map {|x,y| x } # ⇝ [1,3]
```

We also support this idiomatic construct in Switch to facilitate positional access on arrays, and nominal access on records. Since we pass customized objects to blocks in order to assemble expression trees, Ruby does not inherently know how to split them properly into their constituting elements. For this reason, we consider the resulting parse-tree of an expression

e`.`m `{|`x_1`,...,`x_n`|` e_{body} `}` and replace it by e`.`m `{|`x`|` e'_{body} `}` .

In case of arrays, all variables x_1 through x_n that occur free in e_{body} are substituted accordingly by x`[1]` through x`[`n`]` in order to form e'_{body}. Similarly, if e consists of records nominal references are used to access those records by their field names. The type system we devise in the following section will help us to make this distinction.

Idioms enable developers to concisely express well-known programming patterns and additionally may improve the readability of their queries. The idioms listed above are a small fraction of those available in Ruby. Because most of them serve as shortcuts that are instantly converted into core constructs, most of them can smoothly participate in Switch queries.

2.5 The Language

Since SWITCH is a domain specific language that targets the execution on a database back-end, this affects its design in some points. A relational database schema is static and comprised of (1) a finite set of table names, (2) a table schema that links each column with a domain, and (3) a set of integrity constraints, such as *primary* and *foreign keys*. Contrary to this, RUBY is a programming language furnished with concepts such as dynamic typing that inherently contradict the static nature of a database schema. In this section and the following ones, we shall define the expressions that are allowed in SWITCH and are thus suitable for being translated into a database query.

Table 2.1 presents a mapping between the *concrete syntax* (or *surface syntax*) and *abstract syntax*. Whereas the concrete syntax specifies how SWITCH terms are written in RUBY itself, the abstract syntax is used as a notational device to formalize the rules following in the next sections. In the following discussion we use $\bar{e}$ as a shorthand for $e_1, \ldots, e_n$ and $\bar{\iota}$=>$\bar{e}$ for ι_1=>$e_1, \ldots, \iota_n$=>e_n. Likewise, for types, $\bar{\tau}$ is used as an abbreviation for $\tau_1, \ldots, \tau_n$, and $\bar{\sigma}$ for $\sigma_1, \ldots, \sigma_m$ (with $n, m \in \mathbb{N}$).

SWITCH emphasizes the functional facets of RUBY, which we have already identified in Section 2.3.1: We adopt the lenient syntactical and semantical conventions of the host language as well as arrays and hashes, the principal data structures for collections. Within SWITCH they are reinterpreted and poured into a formal framework that on the one hand fits snugly into the RUBY environment and on the other hand plays along with database query processors as target execution engines. Whereas objects are ubiquitous in RUBY, at its core SWITCH is oblivious of object hierarchies and method calls; the information that we actually deal with objects is already lost in the abstract syntax tree. Consequently even the type system reasons in terms of tuples, records and lists rather than objects.

The following list confronts the RUBY idioms with their interpretation in SWITCH:

$\boxed{1}$ **Variables.** Variables are closely tied to the block notation in SWITCH. A block in SWITCH is the only language construct that may introduce a variable scope.

$\boxed{2}$ **Base Types.** SWITCH is equipped with standard base types, such as numbers, strings and booleans. Whereas numbers and strings are inherently supported by database systems, boolean values are neglected by some implementations[7] and hence approximated in SWITCH by integer values 1 (for `true`) and 0 (for `false`).

[7]Most notably IBM DB2, does not consider boolean as a native data type in SQL queries.

Expressions

Item			Abstract Syntax	Concrete Syntax
$\boxed{1}$	e	$:=$	`var[`x`]`	x
$\boxed{2}$		$\mid$	`int[`i`]` $\mid$ `dec[`d`]` $\mid$ `str[s]`	$i \mid d \mid$ `"`s`"`
		$\mid$	`bool[`b`]`	`true` $\mid$ `false`
$\boxed{3}$		$\mid$	`()` $\mid$ `(`$\bar{e}$`)`	`[]` $\mid$ `[`$\bar{e}$`]`
$\boxed{4}$		$\mid$	`{}` $\mid$ `{`$\bar{\iota}$`=>`$\bar{e}$`}` $\mid$ `proj(`e`,`ι`)`	`{}` $\mid$ `{`$\bar{\iota}$`=>`$\bar{e}$`}` $\mid$ e`.`ι
$\boxed{5}$		$\mid$	`<>` $\mid$ `<`$\bar{e}$`>`	–
$\boxed{6}$		$\mid$	`table[R]`	R
$\boxed{7}$		$\mid$	`if(`e_1`,`e_2`,`e_3`)`	`if` e_1 `then` e_2 `else` e_3 `end`
$\boxed{8}$		$\mid$	m`(`e`,`$\bar{e}$`)`	e`.`m`(`$\bar{e}$`)`
		$\mid$	m`(`e`,`$\lambda x.e_b$`)`	e`.`m `{|`x`|` e_b`}`
$\boxed{9}$		$\mid$	$\circledast$`(`e_1, e_2`)` $\mid$ $\olessthan$`(`e_1, e_2`)` $\mid$ $\owedge$`(`e_1, e_2`)`	$e_1 \circledast e_2 \mid e_1 \olessthan e_2 \mid e_1 \owedge e_2$

Meta variables

e	expression	x	variable name
i	literal integer	d	literal decimal
s	literal string	b	literal boolean
ι	field name	m	method name
R	table name	$\circledast$	infix operator ($\in$ {`+`, `-`, `*`, `/`, `%`})
$\olessthan$	infix operator ($\in$ {`==`, `<`, `>`, `<=`, `>=`})	$\owedge$	infix operator ($\in$ {`&`, `|`})

Table 2.1: Switch source language reference

Types

τ, σ, υ	$::=$	$\mathit{Int} \mid \mathit{Dec} \mid \mathit{Bool} \mid \mathit{Str}$	atomic values
	$\mid$	$\forall i \in \mathbb{Z} : \lbag i \rbag$	integer literals
	$\mid$	$() \mid (\bar{\tau})$	tuples
	$\mid$	$[\tau]$	lists
	$\mid$	$\{\} \mid \{\bar{\iota} \colon \bar{\tau}\}$	records
	$\mid$	$\tau \rightarrow \sigma$	functions
	$\mid$	$\tau^{\mathtt{t}_1} \wedge \sigma^{\mathtt{t}_2}$	intersection types (with annotations $\mathtt{t}_1$ and $\mathtt{t}_2$)

Table 2.2: Types in Switch

$\boxed{3}$ **Tuples.** As a consequence of dynamic typing, arrays in RUBY can contain arbitrary types. The expression `a = [1, "Lisa", 23.3]`, for example, is a RUBY array containing an integer, a string and a decimal value. Whereas in RUBY this notation is an idiomatic way to implicitly denote product types, we make this explicit in SWITCH by introducing the tuple

$$(\texttt{int}[1], \texttt{str}[\text{Lisa}], \texttt{dec}[23.3]) \ ,$$

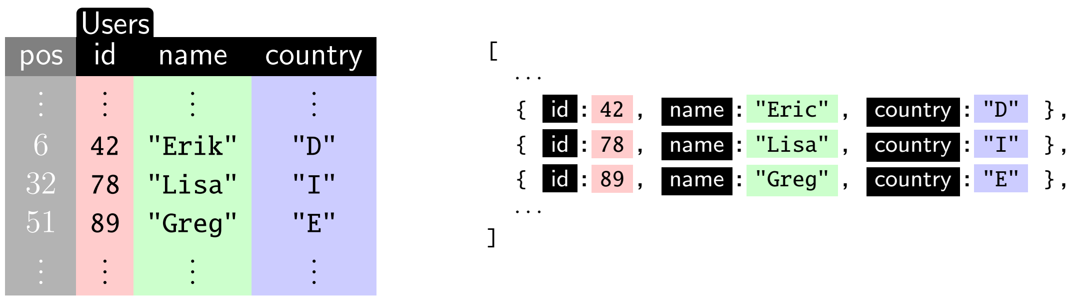
in the abstract syntax tree. The tuple `a` is thus a 3-tuple of type $(\mathit{Int}, \mathit{Str}, \mathit{Dec})$. Likewise `a[2]` denotes the projection on the second field; in general, assuming that `t` is an n-tuple, i must be in the range $\{0, \ldots, n-1\}$.

$\boxed{4}$ **Records.** Hashes are a set of key-value pairs in RUBY, such as the hash `h = {:id=>1,:name=>"Lisa"}`. In this example the keys `:id` and `:name`—instances of the class `Symbol`—are linked with an integer and a string value respectively. In RUBY virtually every object can be used as a hash key. A value that corresponds to a key value is retrieved through square bracket notation as, for example, `h[:name]`.

In SWITCH hashes are regarded as records. The above hash, for example, is typed as $\{\text{id}\colon \mathit{Int}, \text{name}\colon \mathit{Str}\}$. Whereas in RUBY arbitrary objects can be used as keys in hashes, in SWITCH are restricted labels for records to being strings or symbols. Another difference between the RUBY hashes and records in SWITCH is that their fields are indexed differently. In SWITCH a projection on the labels of a record is obtained by a method call on its labels: the `:name`-field in the above hash, for example, is projected by `h.name` in SWITCH, rather than `h[:name]`.

$\boxed{5}$ **Lists.** The list constructor denoted by <$\bar{e}$> describes finite-length lists whose elements evaluate to an equal type; the list <`int[1]`, `int[2]`, `int[3]`>, *e.g.*, has the type $[\mathit{Int}]$. Note that the list constructor does not have a counterpart in the concrete syntax of SWITCH. The rationale behind this is that RUBY arrays, which are interpreted as tuples in SWITCH, can be turned into lists under certain circumstances. SWITCH, however, automatically decides when such a conversion is to take place.

$\boxed{6}$ **Table References.** Each identifier with a capital first letter is recognized as a reference to a table in the backing store. As we mentioned in the preceeding section, the name resolution for constants has been customized to look up tables in the database schema in order to create a table object. A database table in SWITCH is perceived as a list of hashes, as depicted in Figure 2.11. Note that the order in Figure 2.11a is explicitly given by the column `pos`; we shall define how this column evolves in Chapter 3. Furthermore, we refer to the records



	Users		
pos	id	name	country
⋮	⋮	⋮	⋮
6	42	"Erik"	"D"
32	78	"Lisa"	"I"
51	89	"Greg"	"E"
⋮	⋮	⋮	⋮

(a)

```
[
  ...
  { id : 42 , name : "Eric" , country : "D" },
  { id : 78 , name : "Lisa" , country : "I" },
  { id : 89 , name : "Greg" , country : "E" },
  ...
]
```

(b)

Figure 2.11: Tables are perceived as list of records in SWITCH.

enclosed in square brackets in Figure 2.11b as *list* rather than *tuple*; under certain circumstances tuples can be regarded as lists, which shall be explain in more detail when we specify *subtyping* as it is used in SWITCH.

$\boxed{7}$ **Conditionals.** In the same way as in other programming languages, conditionals in SWITCH branch into true and false parts. Depending on the guard e_1, that must evaluate to a boolean expression, the conditional `if(`e_1`,`e_2`,`e_3`)` evaluates to either e_2 for true or e_3 for false. In contrast to vanilla RUBY, the conditional in SWITCH is only valid if and e_3 are typed equally.

$\boxed{8}$ **Method Calls.** A method in RUBY is called by specifying a receiver object and the method name, along with optional parameters and a block:

$$e_1\texttt{.}m\texttt{(}e_2\texttt{,}e_3\texttt{,}\ldots\texttt{,}e_n\texttt{)}\ \texttt{\{|}x\texttt{|}\ e_b\texttt{\}}$$

As SWITCH emphasizes the functional aspects of RUBY, a method call is just a notation for a function call in which the reveiver is bound to its first argument. Likewise, a block `{|`x`|` e_b`}` is construed as an abstraction $\lambda x.e_b$ of a variable x from a term e_b; the scope of x is the body e_b. Within the body e_b a single expression, which must be likewise a valid SWITCH term, is allowed. The above RUBY method call is interpreted by SWITCH as follows:

$$m\texttt{(}e_1\texttt{,}e_2\texttt{,}e_3\texttt{,}\ldots\texttt{,}e_n\texttt{,}\lambda x.e_b\texttt{)}$$

Table 2.3 shows the functions predefined in SWITCH together in combination with their associated types. Note that the type of the first argument is always the type of the receiver. The functions that are tagged with {L, T}, such as **concat**$^{\{\mathsf{L},\mathsf{T}\}}$, are overloaded in SWITCH and have a runtime definition for both tuples and lists.

Built-in functions

$\texttt{concat}^{\{\mathsf{L},\mathsf{T}\}}$	$:: [\tau] \times [\tau] \to [\tau]^{\mathsf{L}}$	concatenation
	$\wedge (\bar{\tau}) \times (\bar{\sigma}) \to (\bar{\tau}, \bar{\sigma})^{\mathsf{T}}$	
$\texttt{[}\cdot\texttt{]}^{\{\mathsf{L},\mathsf{T}\}}$	$:: [\tau] \times \mathit{Int} \to \tau^{\mathsf{L}}$	positional access
	$\wedge (\bar{\tau}) \times \wr i \int \to {\tau_i}^{\mathsf{T}}$ with $i \in \{1, \ldots, n\}$	
$\texttt{first}^{\{\mathsf{L},\mathsf{T}\}}$	$:: [\tau] \to \tau^{\mathsf{L}} \wedge (\bar{\tau}) \to {\tau_1}^{\mathsf{T}}$	first element
$\texttt{last}^{\{\mathsf{L},\mathsf{T}\}}$	$:: [\tau] \to \tau^{\mathsf{L}} \wedge (\bar{\tau}) \to {\tau_n}^{\mathsf{T}}$	last element
$\texttt{take}^{\{\mathsf{L},\mathsf{T}\}}$	$:: [\tau] \times \mathit{Int} \to [\tau]^{\mathsf{L}}$	keep prefix
	$\wedge (\bar{\tau}) \times \wr i \int \to (\tau_1, \ldots, \tau_i)^{\mathsf{T}}$	
$\texttt{drop}^{\{\mathsf{L},\mathsf{T}\}}$	$:: [\tau] \times \mathit{Int} \to [\tau]^{\mathsf{L}}$	keep suffix
	$\wedge (\bar{\tau}) \times \wr i \int \to (\tau_{i+1}, \ldots, \tau_n)^{\mathsf{T}}$	
$\texttt{reverse}^{\{\mathsf{L},\mathsf{T}\}}$	$:: [\tau] \to [\tau]^{\mathsf{L}}$	reversal
	$\wedge (\tau_1, \ldots, \tau_n) \to (\tau_n, \ldots, \tau_1)^{\mathsf{T}}$	
$\texttt{length}^{\{\mathsf{L},\mathsf{T}\}}$	$:: [\tau] \to \mathit{Int}^{\mathsf{L}}$	list length
	$\wedge (\bar{\tau}) \to \wr n \int^{\mathsf{T}}$	
`flatten`	$:: [[\tau]] \to [\tau]$	list flattening
`sum`, `avg`,		
`min`, `max`	$:: [\,\mathrm{num}(\tau)] \to \mathrm{num}\,\tau$	list aggregation
`member?`	$:: [\,\mathrm{atom}(\tau)] \times \mathrm{atom}(\tau) \to \mathit{Bool}$	element lookup
`uniq`	$:: [\,\mathrm{atom}(\tau)] \to [\,\mathrm{atom}(\tau)]$	duplicate elimination
`zip`	$:: ([\tau], [\sigma]) \to [(\tau, \sigma)]$	zip
`unzip`	$:: [(\tau, \sigma)] \to ([\tau], [\sigma])$	unzip

Higher order built-in functions

`map`	$:: [\tau] \times (\tau \to \sigma) \to [\sigma]$	iterate over elements
`select`, `reject`	$:: [\tau] \times (\tau \to \mathit{Bool}) \to [\tau]$	filter elements
`flat_map`	$:: [\tau] \times (\tau \to [\sigma]) \to [\sigma]$	iteration and flattening
`all?`, `any?`	$:: [\tau] \times (\tau \to \mathit{Bool}) \to \mathit{Bool}$	quantification
`take_while`, `drop_while`	$:: [\tau] \times (\tau \to \mathit{Bool}) \to [\tau]$	prefix und suffix
`count`	$:: [\tau] \times (\tau \to \mathit{Bool}) \to \mathit{Int}$	count elements
`sort_by`	$:: [\tau] \times (\tau \to \mathrm{atom}(\sigma)) \to [\tau]$	sorting
`min_by`, `max_by`	$:: [\tau] \times (\tau \to \mathrm{atom}(\sigma)) \to \tau$	minimum and maximum
`group_with`	$:: [\tau] \times (\tau \to \mathrm{atom}(\sigma)) \to [[\tau]]$	grouping
`partition`	$:: [\tau] \times (\tau \to \mathit{Bool}) \to [[\tau]]$	partition

Table 2.3: A library of predefined SWITCH built-in functions and macros.

$\boxed{9}$ **Expressions.** In RUBY there are no primitive values; even numbers and boolean values are ordinary objects with methods. Infix operators, such as `+` and `<`, are no exception: each expression in infix notation is, such as `1 + 2`, is desugared into a method call. The preceding example, *e.g.*, is equivalent to `1.+(2)` and evaluates to `3`.

SWITCH defines ordinary arithmetic expressions as well as comparison operators on numbers, strings and booleans. Because the RUBY equivalents for logical conjunction (`&&`) and disjunction (`||`) are not defined as customizable methods, we employ the bitwise operators (`!` and `&`) to assume their role. Together with the negation operator (`!`) they complete the set of logical connectives in SWITCH.

2.6 A Type System for SWITCH

The type system we will describe in the current section forms the backbone of SWITCH. Its purpose is to decide for an arbitrary expression whether it is a valid term in SWITCH. Consequently a term that passes the type system is well behaved and suitable to be translated into a representation in relational algebra and thus into a SQL query executable on a database back-end. Moreover, the type system establishes the foundation for several type sensitive rewrites, such as converting tuples into lists.

The type environment Γ is a partial function that maps variable names in $\mathcal{V}$ to their types in $\mathcal{T}$:

$$\Gamma\colon \mathcal{V} \to \mathcal{T} \qquad \text{with} \qquad (\Gamma, v :: \tau)\, x = \begin{cases} \tau & \text{if } v = x \\ \Gamma\, x & \text{else} \end{cases}$$

The type of a variable x is thus retrieved by applying it to the environment $\tau = \Gamma\, x$. Updating of the type environment is denoted by $\Gamma, x :: \tau$. The empty environment is simply a function that returns $\bot$ for any argument applied to it. The type environment is initialized with the functions in Table 2.3, so that it turns into a dictionary in which the type for any given function name can be looked up: the type of a variable name and a method type are retrieved equivalently.

A type rule as written in Table 2.4 has the general shape

$$\Gamma \vdash e :: \tau$$

and is read as “In type environment Γ the expression e is typed as τ.”

$$\frac{}{\Gamma \vdash \texttt{int[}i\texttt{]} :: \rmoustache i \lmoustache}\,(\textsc{Ty-Int}) \qquad \frac{}{\Gamma \vdash \texttt{dec[}d\texttt{]} :: \mathit{Dec}}\,(\textsc{Ty-Dec}) \qquad \frac{}{\Gamma \vdash \texttt{str[}\mathrm{s}\texttt{]} :: \mathit{Str}}\,(\textsc{Ty-Str})$$

$$\frac{}{\Gamma \vdash \texttt{bool[}b\texttt{]} :: \mathit{Bool}}\,(\textsc{Ty-Bool}) \qquad \frac{\Gamma\, x = \tau}{\Gamma \vdash \texttt{var[}x\texttt{]} :: \tau}\,(\textsc{Ty-Var})$$

$$\frac{}{\Gamma \vdash \texttt{()} :: ()}\,(\textsc{Ty-TupEmpty}) \qquad \frac{\Gamma \vdash e_i :: \tau_i \,\Big|_{i=1,\ldots,n}}{\Gamma \vdash \texttt{(}\bar{e}\texttt{)} :: (\bar{\tau})}\,(\textsc{Ty-TupCons})$$

$$\frac{}{\Gamma \vdash \texttt{\{\}} :: \{\}}\,(\textsc{Ty-RecEmpty}) \qquad \frac{\Gamma \vdash e_i :: \tau_i \,\Big|_{i=1,\ldots,n}}{\Gamma \vdash \texttt{\{}\bar{\iota}\texttt{=>}\bar{e}\texttt{\}} :: \{\bar{\iota}\colon \bar{\tau}\}}\,(\textsc{Ty-RecCons})$$

$$\frac{}{\Gamma \vdash \texttt{<>} :: [\tau]}\,(\textsc{Ty-ListEmpty}) \qquad \frac{\Gamma \vdash e_i :: \tau \,\Big|_{i=1,\ldots,n}}{\Gamma \vdash \texttt{<}\bar{e}\texttt{>} :: [\tau]}\,(\textsc{Ty-ListCons})$$

$$\frac{\Gamma \vdash e :: \{\ldots, \iota\colon \tau, \ldots\}}{\Gamma \vdash \texttt{proj(}e\texttt{,}\iota\texttt{)} :: \tau}\,(\textsc{Ty-RecProj}) \qquad \frac{\text{reflection on the database schema}}{\Gamma \vdash \texttt{table[R]} :: [\{\bar{\iota}\colon \bar{\tau}\}]}\,(\textsc{Ty-TableRef})$$

$$\frac{\Gamma \vdash e_1 :: \mathit{Bool} \qquad \Gamma \vdash e_i :: \tau \,\Big|_{i=2,3}}{\Gamma \vdash \texttt{if(}e_1\texttt{,}e_2\texttt{,}e_3\texttt{)} :: \tau}\,(\textsc{Ty-If}) \qquad \frac{\mathrm{num}(\tau) \qquad \Gamma \vdash e_i :: \tau \,\Big|_{i=1,2}}{\Gamma \vdash \circledast\texttt{(}e_1, e_2\texttt{)} :: \tau}\,(\textsc{Ty-Arith})$$

$$\frac{\mathrm{atom}(\tau) \qquad \Gamma \vdash e_i :: \tau \,\Big|_{i=1,2}}{\Gamma \vdash \olessthan\texttt{(}e_1, e_2\texttt{)} :: \mathit{Bool}}\,(\textsc{Ty-Comp}) \qquad \frac{\Gamma \vdash e_i :: \mathit{Bool} \,\Big|_{i=1,2}}{\Gamma \vdash \owedge\texttt{(}e_1, e_2\texttt{)} :: \mathit{Bool}}\,(\textsc{Ty-Junc})$$

$$\frac{\begin{array}{c}\Gamma \vdash m :: \tau_1 \times \ldots \times \tau_n \to \sigma \\ \Gamma \vdash e_i :: \tau_i \,\Big|_{i=1,\ldots,n}\end{array}}{\Gamma \vdash m\texttt{(}\bar{e}\texttt{)} :: \sigma}\,(\textsc{Ty-Func}) \qquad \frac{\begin{array}{c}\Gamma \vdash m :: [\tau] \times (\tau \to \tau_b) \to \sigma \\ \Gamma \vdash e :: [\tau] \quad \Gamma, x :: \tau \vdash e_b :: \tau_b\end{array}}{\Gamma \vdash m\texttt{(}e, \lambda x.e_b\texttt{)} :: \sigma}\,(\textsc{Ty-FuncBlock})$$

$$\frac{\Gamma \vdash e :: \sigma \qquad \sigma \lessdot \tau}{\Gamma \vdash e :: \tau}\,(\textsc{Ty-Sub})$$

Table 2.4: Type system of SWITCH

We define the typing relation as a set of inference rules summarized in Table 2.4. In the following we will provide a description for each rule:

Whereas the rules (Ty-Dec) and (Ty-Str) simply state the types for decimal numbers and strings, the rule for (Ty-Int) deserves more attention: As the functions for positional access (`[·]`$^{\{\mathsf{L},\mathsf{T}\}}$), keep prefix (`take`$^{\{\mathsf{L},\mathsf{T}\}}$), and keep suffix (`drop`$^{\{\mathsf{L},\mathsf{T}\}}$) work on lists as well as on tuples, in the latter case the result type is defined in terms of the integer value i, whose type is determined by the second argument of this function. Hence, the value i must be available when the type system is determining the result type of such functions. The type $\lceil i \rfloor$ is a *dependent type* [XP99] that depends on the value of an integer literal. Given that only literal integers (`int[`i`]`) are typed as $\lceil i \rfloor$, we can be sure that i is to be involved in the reasoning process for the type system. The literal `int[4]`, for example, has the type $\lceil \mathtt{4} \rfloor$. Each dependent expression $e :: \lceil i \rfloor$ can easily be converted into an integer expression; we will come back to this when we describe the subtype relation.

A literal array in Ruby, such as `[`e_1`,`$\ldots$`,`e_n`]`, leads to a tuple in Switch. Consequently, the rules (Ty-TupEmpty) and (Ty-TupCons) capture the empty tuple `()` and composite tuples `(`$e_1, \ldots, e_n$`)` respectively. Not surprisingly, the type of composite tuples depends on its constituents. The rules (Ty-RecEmpty) and (Ty-RecCons), capturing the empty `{}` and composite records `{`ι_1`=>`$e_1, \ldots, \iota_n$`=>`e_n`}`, work analogously to the rules for tuples. However, we insist that all labels ι_1 to ι_n in a record term or type are distinct. This prevents ambiguities because the rule (Ty-RecProj) draws the type for the projection `proj(`e`,`ι`)` on the field labeled by ι directly from the type of the record e.

The rule (Ty-TableRef) is meant to be understood as follows: `table[R]`, which embodies a table in the underlying database, is typed as a list of records. Formally a table can be interpreted as a *bag of tuples*. Lists, as well as bags (or multisets), require their constituents to be equally typed. We justify the representation of an unordered bag in terms of an ordered list by introducing an order based on one or a combination of attributes of the table. Even though the actual choice is irrelevant, usually the primary key is selected. The primary key presents a reasonable choice because it is usually closely tied to the physical organization of a table (*clustered index*). The record labels are obtained by reflection on the database schema; each field in a tuple is named according to the table schema. Likewise, the atomic types of the tuple fields are extracted from the schema and directly mapped to their atomic equivalents in Switch.

Rule (Ty-If) assigns a type to a conditional expression based on the types of its subexpressions. The guard e_1 must evaluate to a boolean, whereas both e_2 and e_3 must evaluate to the same type.

The rules for arithmetic (TY-ARITH) and boolean expressions (TY-COMP) depend on their subexpressions. In (TY-ARITH) the type of e_1 and e_2 is required to be numeric whereas (TY-COMP) only demands their types to be atomic. The boolean connectives "and" (`&`) and "or" (`|`), captured in rule (TY-JUNC), both evaluate to a boolean.

The rule for typing function calls (TY-FUNC) insists that the types of their arguments exactly match their domain types. Quite a few functions in SWITCH are *overloaded*, and work on both lists and tuples. Based on the current rules the type system is annoyingly rigid and rejects any function call that involves one of these functions because it can not decide whether the variant for lists or tuples is meant. However, in the next section we will refine the rules and thereby enable that the type system to make an educated choice between the list and the tuple variant.

All functions accepting a block as their argument share the feature that of performing some sort of iteration over a list of elements. In the course of this iteration, the block is successively applied to each element in the list. Each time the body e_b is therefore evaluated in the context of one of these items. Fortunatly, when typing such an expression we may rely on the fact that all the elements of a list of types $[\tau]$ have the common type τ: the type of the body e_b may thus be derived under the assumption that x is of type τ; this is formalized by rule (TY-FUNCBLOCK).

We conclude the description of the type system with the rule (TY-SUB), which links the type system to the subtype relation we shall define in the next section. The rule says that if $\sigma \lessdot \tau$, an element e of type σ is likewise an element of type τ.

2.7 A Horizontal and Vertical Array-Representation

The role of arrays in RUBY is somewhat ambiguous: As RUBY is based on dynamic typing, it does not have notationally distinct forms for tuples and lists. On the one hand they are used as an idiom to denote tuples, on the other hand they represent collections of decent size, in which all their elements have an equal type. In the latter case their intention clearly is to act as list structures.

When working with database systems, this behavior becomes problematic because tuples and lists are represented differently in a database table. A RUBY array, such as `[1, "Lisa", 23.3]`, has the product type $(\mathit{Int}, \mathit{Str}, \mathit{Dec})$ in SWITCH. It is likewise rendered into a tuple in a table, as illustrated in Figure 2.12; we refer to this as *vertical representation*.

With the decision to type literal arrays as product types, even the RUBY array `[1,2,3]` is perceived as a tuple and hence typed as $(\mathit{Int}, \mathit{Int}, \mathit{Int})$. This leads

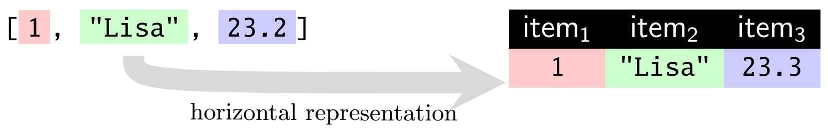


Figure 2.12: Horizontal representation of an array in a database. Order is implicitly given by the position of the columns.

the compiler to reject the following expression that, to the RUBY programmer, is seemingly well behaved:

```
[1, 2, 3].map {|x| x + 1} # ⇝ [2, 3, 4]
```

The function `map`, which has the type $[\tau] \times (\tau \to \sigma) \to [\sigma]$, expects a list as first argument. Because the `[1,2,3]` is a tuple, the types do not match, and the expression is not typeable. According to the present type system, `map()` may only be applied to terms that either directly represent a table, such as the term R, or are derived from a table.

The above tuple `[1,2,3]` could be easily transferred into a list representation: As all elements have a common type they could be rendered into a single column rather than using three columns (Figure 2.13). The position of the elements in the tuple are preserved by providing an additional `pos` column. In contrast to the represenation for tuples, this is referred to as *vertical representation*.

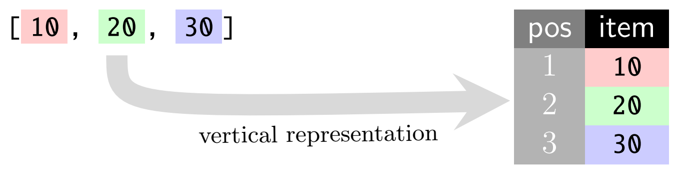


Figure 2.13: Vertical representation of an array in a table. Order is explicitly given by a further column `pos`.

This conversion comes at a cost: we lose positional access in constant time, the key characteristic of arrays. In the horizontal representation in form of tuples, positional access is preserved by a simple projection on a table attribute that is derived at compile time. In the vertical representation however positional access is obtained by a comparison with the `pos` column at runtime, which, at best, is achievable in logarithmic time depending on the cardinality of the table.

The second problem that we tackle in the subtype relation is to derive the proper type for the overloaded functions we presented in Table 2.3. Each function that is tagged with $\{\mathsf{L}, \mathsf{T}\}$ can be applied to both tuples and lists. Because lists and tuple are, due to their representation, treated differently at runtime, we must exactly know which implementation is used. The subtype relation lays the foundation for this decision and is used in *type-coercion* process we shall describe in the next section. In the following discussion we use the function `concat` as representative to examine how overloading of functions is implemented in SWITCH.

In RUBY, `concat`, appends the lists in the argument to the caller, which is likewise an array:

```
[1, "a", 2].concat(["b", 3]) # ⇝ [1, "a", 2, "b", 3]
```

In SWITCH, however, the type of `concat()` is overloaded and provides an implementation for the case that both arguments are lists and another one the case that both arguments are tuples. In the example mentioned both arguments evaluate to tuples so the type that fits best for this example is $(\bar{\tau}) \times (\bar{\sigma}) \to (\bar{\tau}, \bar{\sigma})$, which is instantiated as

$$(\mathit{Int}, \mathit{Str}, \mathit{Int}) \times (\mathit{Str}, \mathit{Int}) \to (\mathit{Int}, \mathit{Str}, \mathit{Int}, \mathit{Str}, \mathit{Int})$$

For RUBY programmers, the following expression, which involves a table access to `Users`, is also well behaved:

$$\begin{array}{l}\texttt{Users.map \{|x| x.id\}.} \\ \quad\quad \texttt{concat([23, 42]) \# } \rightsquigarrow \texttt{ [}\underbrace{\texttt{..., ..., ...}}_{\texttt{result from map}}\texttt{, 23, 42]} \end{array} \qquad (Q_1)$$

In this case, `map()` evaluates to a list whose elements are typed as integer—assuming that `x.id` evaluates to an integer—and hence the function `concat()` is confronted with a list as its first argument. The second argument (`[23, 42]`), however is a tuple and does not conform with the type of the second parameter of $([\tau] \times [\tau] \to [\tau])$. Fortunatly we can promote this tuple to a list of the type integer and eventually apply the version of `concat()` that is concerned with lists rather than tuples; its type may be instantiated as follows:

$$[\mathit{Int}] \times [\mathit{Int}] \to [\mathit{Int}]$$

2.7.1 Formalization via Subtyping

The subtype relation in Table 2.5 refines the typing rules of Table 2.4, so that they implement the concepts explained above. The subtype relation has the general shape

$$\tau \lessdot \sigma$$

and is pronounced as “A term of type τ can safely be used in a context where a term of type σ is expected.” Subtyping is widely known as the *principle of safe substitution.*

$$\frac{}{\tau \lessdot \tau}\,\text{(SubTy-Refl)} \qquad \frac{\tau \lessdot \upsilon \qquad \upsilon \lessdot \sigma}{\tau \lessdot \sigma}\,\text{(SubTy-Trans)}$$

$$\frac{}{\mathit{Int} \lessdot \mathit{Dec}}\,\text{(SubTy-Int)} \qquad \frac{}{\lbag i \rbag \lessdot \mathit{Int}}\,\text{(SubTy-Num)}$$

$$\frac{\tau_i \lessdot \sigma_i \big|_{i=1,\ldots,n}}{\{\bar{\iota} \colon \bar{\tau}\} \lessdot \{\bar{\iota} \colon \bar{\sigma}\}}\,\text{(SubTy-Rec)} \qquad \frac{\tau \lessdot \sigma}{[\tau] \lessdot [\sigma]}\,\text{(SubTy-List)}$$

$$\frac{\tau_i \lessdot \sigma_i \big|_{i=1,\ldots,n}}{(\bar{\tau}) \lessdot (\bar{\sigma})}\,\text{(SubTy-Tup)} \qquad \frac{\tau_i \lessdot \sigma \big|_{i=1,\ldots,n}}{(\bar{\tau}) \lessdot [\sigma]}\,\text{(SubTy-Tup-List)}$$

$$\frac{}{\tau^{\mathsf{t}_1} \wedge \sigma^{\mathsf{t}_2} \lessdot \tau^{\mathsf{t}_1}}\,\text{(SubTy-Inter1)} \qquad \frac{}{\tau^{\mathsf{t}_1} \wedge \sigma^{\mathsf{t}_2} \lessdot \sigma^{\mathsf{t}_2}}\,\text{(SubTy-Inter2)}$$

Table 2.5: Subtypes in Switch

We will consider each form of type that is used in Switch separately. For each one we introduce one or more rules formalizing under which circumstances it is safe to substitute one type for another.

First, we stipulate the more general properties of the subtype relation: the rules (SubTy-Refl) and (SubTy-Trans) implement reflexivity and transitivity respectively. A situation in which equal types are subtypes of one another is captured by reflexivity. Transitivity, denominates the case in which a type τ is indirectly—through the type υ—a subtype of another type σ.

The rule (SubTy-Int) states that an integer term can be used whenever a decimal term is expected. Similarly, a term of the dependent type $\lbag i \rbag$ can be promoted to a term of type integer.

Rule (SubTy-Rec) applies to record types whose labels are identical. Additionally, it is safe to allow the types of individual fields to vary as long as this is captured by the subtype relation. The same applies to product types, which are captured by the rule (SUBTY-TUP).

A list whose elements are of type τ could safely be regarded as a list with elements of type σ; rule (SUBTY-LIST) captures this situation.

Rule (SUBTY-TUP-LIST) formalizes the intuition that a tuple whose elements evaluate to a common type σ can be safely promoted to a list of type σ. The rule generalizes this observation allowing the individual tuple types to vary, provided that they are a subtype of the list type σ.

The overloading we implemented in SWITCH is a simple form of parametric polymorphism called *finitary polymorphism* [Pie91]. In this regard, we enriched the inhabitants of our type system with a new type $\tau^{\mathsf{t}_1} \wedge \sigma^{\mathsf{t}_2}$ for every pair of types τ and σ. This new type is meant to contain all terms that belong to the type τ as well as to the type σ, or in other words, an element of the intersection $\tau^{\mathsf{t}_1} \wedge \sigma^{\mathsf{t}_2}$ is furnished with enough information to coerce it to be either an element of τ, which is captured by rule (SUBTY-INTER1), or an element of σ, reflected by rule (SUBTY-INTER2). Intersection types serve a purpose similar to method overloading in JAVA [Jav], although they are resolved at runtime via type introspection rather than at compile time via type checking. For example, the funcion **`concat()`** in SWITCH is overloaded to work over both lists and tuples:

$$\texttt{concat} :: [\tau] \times [\tau] \to [\tau] \qquad \text{and} \qquad \texttt{concat} :: (\bar{\tau}) \times (\bar{\sigma}) \to (\bar{\tau}, \bar{\sigma})$$

In SWITCH we use the intersection type to provide the following “combined type” that can assume either the list type or tuple type. For the above function this leads to:

$$\texttt{concat}^{\{\mathsf{L},\mathsf{T}\}} :: [\tau] \times [\tau] \to [\tau]^{\mathsf{L}} \wedge (\bar{\tau}) \times (\bar{\sigma}) \to (\bar{\tau}, \bar{\sigma})^{\mathsf{T}}$$

The semantics for both functions are similar. On the one hand, for the list-typed version of **`concat`** the application on two lists delivers the following result (due to readability, we write **`1`** instead of **`int[1]`** for literals):

$$\texttt{concat}^{\mathsf{L}}\texttt{(<1,2>,<3>)} \rightsquigarrow \texttt{<1,2,3>}$$

On the other hand, the tuple-typed version exposes a similar behavior when applied on two tuples with elements of varying types:

$$\texttt{concat}^{\mathsf{T}}\texttt{((1,2),(3.0))} \rightsquigarrow \texttt{(1,2,3.0)}$$

The tags t_1 and t_2 are a peculiarity of the intersection types we propose in this work. They will be used when we get rid of the subtypes in the next step (Section 2.8) by replacing each rule by runtime coercions.

For functions, such as `concat`, the type system can now automatically infer the proper type by making use of either rule (SubTy-Inter1) or rule (SubTy-Inter2) as we demonstrate in the following example (see Table 2.6), in which we derive the type of Query Q_1 on page 38.

2.8 Coerceing Tuples into Lists

In the last section, we expressed the intuition that we can regard tuples as lists under certain circumstances, and we additionally equipped Switch with a simple overloading mechanism. However, the typing mechanism only tells us whether a Switch expression is valid or not. In this section, we will devise a translation between Switch terms that replaces subtyping by compile-time rewrites. If, for example, a tuple is promoted to a list, we gather the individual elements of the tuple and create a literal list at compile time. The result of this operation is reflected on the database back-end where the list is arranged into a column rather than into a tuple within a table.

To rewrite a Switch expression based on the typing information we exactly need to know where subtyping takes place in the derivation tree that is produced while inferring the type of an expression. Consequently we provide functions that are applied to the derivation trees [BtC+91] of the subtype and the type relations respectively. In the rules summarized in Table 2.7 we use $\mathcal{C} \therefore \tau \lessdot \sigma$ to mean a derivation tree $\mathcal{C}$ whose conclusion is a subtyping statement $\tau \lessdot \sigma$. Likewise we use $\mathcal{D} \therefore \Gamma \vdash e :: \tau$ for the derivation tree $\mathcal{D}$ that concludes with $\Gamma \vdash e :: \tau$ in Table 2.9.

2.8.1 Coercion on Subtypes

To rewrite a type based on the subtype relation we use the notation $[\![\cdot]\!]_{\lessdot}$ that, given a derivation tree $\mathcal{C}$ for a subtype statement $\tau \lessdot \sigma$, derives a function that tells us how to rewrite a term so that subtyping for this particular term becomes obsolete. Consequently, $[\![\mathcal{C}]\!]_{\lessdot}$ is a function that translates a Switch term into another Switch term. Note that in the function (CoeSubTy-Tup-List) we employ the list literal <e_1, ... ,e_n> that is only available in the abstract syntax of Switch. So far, the single statement in the surface syntax of Switch that produces a list was table-reference operator via.

Whereas for reflexivity captured in function (CoeSubTy-Trans), we simply rewrite the term by means of the identity function, in function (CoeSubTy-Trans) we compose the rewrite functions—with $f \circ g \equiv \lambda x. f(g(x)))$—that we derived over the derivation trees of its premises into a single function, to which the term t is then applied.

$$\texttt{concat(}\overbrace{\texttt{map(}\underbrace{\texttt{table[Users]}}_{❶}\texttt{,}\lambda x.\underbrace{\texttt{proj(var[}x\texttt{],id)}}_{❷}\texttt{)}}^{❸}\texttt{,}\underbrace{\texttt{(int[23],int[42])}}_{❹}\texttt{)}$$

$$◇1\quad \frac{\text{reflection on database schema}}{\Gamma \vdash ❶ :: [\{\ldots, \texttt{id}\colon \mathit{Int}, \ldots\}]}\,(\textsc{Ty-TableRef})$$

$$◇2\quad \frac{\text{with } \sigma = \{\ldots, \texttt{id}\colon \mathit{Int}, \ldots\} \qquad \dfrac{(\Gamma, x :: \sigma)\, x = \sigma}{\Gamma, x :: \sigma \vdash \texttt{var[}x\texttt{]} :: \sigma}\,(\textsc{Ty-Var})}{\Gamma, x :: \sigma \vdash ❷ :: \mathit{Int}}\,(\textsc{Ty-RecProj})$$

$$◇3\quad \frac{\begin{array}{l}\text{with } \tau = \{\ldots, \texttt{id}\colon \mathit{Int}, \ldots\} \text{ and } \sigma = \mathit{Int}\\ \Gamma \vdash \texttt{map} :: [\tau] \times (\tau \to \sigma) \to [\sigma]\\ \qquad ◇1 \qquad\qquad\qquad\qquad ◇2\end{array}}{\Gamma \vdash ❸ :: [\mathit{Int}]}\,(\textsc{Ty-FuncBlock})$$

$$◇4\quad \frac{\dfrac{}{\Gamma \vdash \texttt{int[23]} :: \wr 23 \int}\,(\textsc{Ty-Int}) \qquad \dfrac{}{\Gamma \vdash \texttt{int[42]} :: \wr 42 \int}\,(\textsc{Ty-Int})}{\Gamma \vdash ❹ :: (\wr 23 \int, \wr 42 \int)}\,(\textsc{Ty-TupCons})$$

$$◇5\quad \frac{\dfrac{}{\wr 23 \int \lessdot \mathit{Int}}\,(\textsc{SubTy-Num}) \qquad \dfrac{}{\wr 42 \int \lessdot \mathit{Int}}\,(\textsc{SubTy-Num})}{(\wr 23 \int, \wr 42 \int) \lessdot [\mathit{Int}]}\,(\textsc{SubTy-Tup-List})$$

$$◇6\quad \frac{◇4 \qquad ◇5}{\Gamma \vdash ❹ :: [\mathit{Int}]}\,(\textsc{Ty-Sub})$$

$$◇7\quad \frac{\begin{array}{l}\text{with } \upsilon = [\mathit{Int}] \times [\mathit{Int}] \to [\mathit{Int}] \text{ and } \delta = (\bar{\tau}) \times (\bar{\sigma}) \to (\bar{\tau}, \bar{\sigma})\\ \Gamma \vdash \texttt{concat} :: \upsilon^{\mathsf{L}} \wedge \delta^{\mathsf{T}} \qquad \dfrac{}{\upsilon^{\mathsf{L}} \wedge \delta^{\mathsf{T}} \lessdot \upsilon^{\mathsf{L}}}\,(\textsc{SubTy-Inter1})\end{array}}{\Gamma \vdash \texttt{concat} :: \upsilon^{\mathsf{L}}}\,(\textsc{Ty-Sub})$$

$$◇★\quad \frac{◇3 \qquad ◇6 \qquad ◇7}{\Gamma \vdash \texttt{concat(}❸\texttt{,}❹\texttt{)} :: [\mathit{Int}]}\,(\textsc{Ty-Func})$$

Table 2.6: Type inference on Query Q_1. In the above inference, we use the abstract syntax `concat(map(table[Users],`$\lambda x.$`proj(`x`,id)),(23,42))`. The type environment starts with $\Gamma = \emptyset$. The rule annotated with ◇★ shows the result type of the expression. Note how subtyping is used to regard the tuple (`(int[23],int[42])`) as list in order to be aligned with the type of the result of `map()`.

The next two functions (CoeSubTy-Int) and (CoeSubTy-Num) turn an integer into a decimal number, and a term that belongs to a dependent type, which was introduced above as a type for literal integers, into an integer respectively.

Function (CoeSubTy-Rec) rewrites a record in a way that in any individual field the corresponding rewrite function $[\![\mathcal{C}_i]\!]_{\lessdot}$ for $i = 1, \dots, n$ is applied. For tuples captured in (CoeSubTy-Tup) we likewise apply the rewrite function to any individual field in order to compose a new tuple where subtyping is no longer needed.

Subtyping in lists occurs when the elements of a list can be coerced into elements of another type. Consequently we have to apply the rewrite function $[\![\mathcal{C}]\!]_{\lessdot}$ on the elements of the list rather than on the list itself. Unfortunately, this leads to an iteration over the list elements during runtime; the coercion is applied to the block variable of `map` that is successively bound to each element of the list.

If we regard a tuple as a function, we again need to consider its constituents. Each individual field of the tuple that belongs to the type τ_i, for each $i = 1, \dots, n$, may be coerced into an element of the list type σ. As a result we access each single tuple element and then coerce it by its corresponding function $[\![\mathcal{C}_i]\!]_{\lessdot}$. Finally, we assemble a list literal, enclosed by <>, whose constitutes are the coerced tuple elements; we express this in function (CoeSubTy-Tup-List).

The last two rules (CoeSubTy-Inter1) and (CoeSubTy-Inter2) apply to derivation trees whose conclusion contains an intersection type. For overloaded functions in SWITCH, whose type definition contain intersection types we provide different compilation rules for the relational target language that reflect the respective table representation of their arguments: horizontal or vertical representation. Consequently we need to pick either `concat`$^{\mathsf{L}}$`()` and `concat`$^{\mathsf{T}}$`()`, that work on list and tuples respectively.

Derived by rule (CoeSubTy-Inter1) and rule (CoeSubTy-Inter2), the purpose of these coercions is to pick a concrete instance of an overloaded function. The tags $\mathtt{t}_1$ or $\mathtt{t}_2$, which are tied to each constituent of an intersection type $\tau^{\mathtt{t}_1} \wedge \sigma^{\mathtt{t}_2}$. are used to label the functions. In the case of `concat()`, this leads to either `concat`$^{\mathsf{L}}$`()` or `concat`$^{\mathsf{T}}$`()`.

Table 2.7: Coercion over the proof tree of the subtypes

$$\left[\!\left[\frac{}{\tau \lessdot \tau}\,(\textsc{SubTy-Refl}) \right]\!\right]_{\lessdot} = \lambda t.t \quad (\textsc{CoeSubTy-Refl})$$

$$\left[\!\left[\frac{\mathcal{C}_1 \therefore\ \tau \lessdot \upsilon \qquad \mathcal{C}_2 \therefore\ \upsilon \lessdot \sigma}{\tau \lessdot \sigma}\,(\textsc{SubTy-Trans}) \right]\!\right]_{\lessdot} = \lambda t.([\![\mathcal{C}_1]\!]_{\lessdot} \circ [\![\mathcal{C}_2]\!]_{\lessdot})\ t \quad (\textsc{CoeSubTy-Trans})$$

$$\left[\!\left[\frac{}{\mathit{Int} \lessdot \mathit{Dec}}\,(\textsc{SubTy-Int}) \right]\!\right]_{\lessdot} = \lambda t.t \quad (\textsc{CoeSubTy-Int}) \qquad \left[\!\left[\frac{}{\wr i \int \lessdot \mathit{Int}}\,(\textsc{SubTy-Num}) \right]\!\right]_{\lessdot} = \lambda t.\mathtt{int[}t\mathtt{]} \quad (\textsc{CoeSubTy-Num})$$

$$\left[\!\left[\frac{\mathcal{C}_i \therefore\ \tau_i \lessdot \sigma_i \big|_{i=1,\ldots,n}}{\{\bar{\iota}\colon \bar{\tau}\} \lessdot \{\bar{\iota}\colon \bar{\sigma}\}}\,(\textsc{SubTy-Rec}) \right]\!\right]_{\lessdot} = \lambda t.\ \mathtt{\{}\iota_1\mathtt{=>}[\![\mathcal{C}_1]\!]_{\lessdot}\,\mathtt{proj(}t\mathtt{,}\iota_1\mathtt{)}, \ldots, \iota_n\mathtt{=>}[\![\mathcal{C}_n]\!]_{\lessdot}\,\mathtt{proj(}t\mathtt{,}\iota_n\mathtt{)\}} \quad (\textsc{CoeSubTy-Rec})$$

$$\left[\!\left[\frac{\mathcal{C}_i \therefore\ \tau_i \lessdot \sigma_i \big|_{i=1,\ldots,n}}{(\bar{\tau}) \lessdot (\bar{\sigma})}\,(\textsc{SubTy-Tup}) \right]\!\right]_{\lessdot} = \lambda t.\mathtt{(}[\![\mathcal{C}_1]\!]_{\lessdot}\,\mathtt{[\cdot]}^{\mathsf{T}}\mathtt{(}t\mathtt{,1)}, \ldots, [\![\mathcal{C}_n]\!]_{\lessdot}\,\mathtt{[\cdot]}^{\mathsf{T}}\mathtt{(}t\mathtt{,}n\mathtt{))} \quad (\textsc{CoeSubTy-Tup})$$

continued on next page

$$\left[\!\!\left[\frac{\mathcal{C} \therefore\ \tau \lessdot \sigma}{[\tau] \lessdot [\sigma]}\textsc{(SubTy-List)}\right]\!\!\right]_{\lessdot} = \lambda t.\texttt{map(}t\texttt{,}\lambda x.[\![\mathcal{C}]\!]_{\lessdot}\ x\texttt{)} \quad \textsc{(CoeSubTy-List)}$$

$$\left[\!\!\left[\frac{\mathcal{C}_i \therefore\ \tau_i \lessdot \sigma\,\big|_{i=1,\dots,n}}{(\bar{\tau}) \lessdot [\sigma]}\textsc{(SubTy-Tup-List)}\right]\!\!\right]_{\lessdot} = \lambda t.\texttt{<}[\![\mathcal{C}_1]\!]_{\lessdot}\ \texttt{[}\cdot\texttt{]}^{\mathsf{T}}\texttt{(}t\texttt{,}1\texttt{)}, \dots, [\![\mathcal{C}_n]\!]_{\lessdot}\ \texttt{[}\cdot\texttt{]}^{\mathsf{T}}\texttt{(}t\texttt{,}n\texttt{)>} \quad \textsc{(CoeSubTy-Tup-List)}$$

$$\left[\!\!\left[\frac{}{\tau^{\mathsf{t}_1} \wedge \sigma^{\mathsf{t}_2} \lessdot \tau^{\mathsf{t}_1}}\textsc{(SubTy-Inter1)}\right]\!\!\right]_{\lessdot} = \lambda t.t^{\mathsf{t}_1} \quad \textsc{(CoeSubTy-Inter1)}$$

$$\left[\!\!\left[\frac{}{\tau^{\mathsf{t}_1} \wedge \sigma^{\mathsf{t}_2} \lessdot \sigma^{\mathsf{t}_2}}\textsc{(SubTy-Inter2)}\right]\!\!\right]_{\lessdot} = \lambda t.t^{\mathsf{t}_2} \quad \textsc{(CoeSubTy-Inter2)}$$

2.8.2 Coercion on Types

In a way similar to that in the preceding section, in which we introduced runtime coercions based on the subtype relation, we will now devise the coercion function operating on derivation trees of conclusions in the type system. However, we use the notation $[\![\cdot]\!]_{::}$ to denote the coercion function that, given a derivation tree $\mathcal{D}$ for a type statement $\Gamma \vdash e :: \tau$ about the expression e, derives a new SWITCH term $[\![\mathcal{D}]\!]_{::}$. Rather than deriving a function that rewrites a term, this time we directly transform the term so that the result of $[\![\mathcal{D}]\!]_{::}$ is again a SWITCH term. As in the preceding section we must consider each shape of a type statement. Table 2.8 exemplifies how the coercion rules elegantly interact with the type inference.

Most of the functions summarized in Table 2.9 are straightforward, such as the coerce functions defined on rules rules regarding the base types in (CoeTy-Int), (CoeTy-Dec) and (CoeTy-Str). In the new expression we simply adopt the literal in the type statement to the left-hand side. The same applies to the function (CoeTy-Var), in which the variable on the left-hand side is merely adopted to the right-hand side of the function.

Whenever we encounter either an empty tuple, record or list in a type statement we immediately build an empty structure of the same type as the result of the functions (CoeTy-TupEmpty), (CoeTy-RecEmpty) and (CoeTy-ListEmpty).

The functions that capture the rules on composite structures like tuples, records and literal lists in (CoeTy-TupCons), (CoeTy-RecCons) and (CoeTy-ListCons) respectively, propagate the rewrite function through the entire expression tree by applying the individual coercion function $[\![\mathcal{D}_i]\!]_{::}$ to the constitutes e_i for each $i = 1, \ldots, n$.

In contrast to the table reference found on the left-hand side of function (CoeTy-Tab), which is merely adopted on the right-hand side, the functions capturing the projection in (CoeTy-RecProj) as well as the functions applying to the conditional in function (CoeTy-If), simply propagate the coerce functions to their corresponding constituents. This also applies to infix operators in (CoeTy-Arith), (CoeTy-Comp) and (CoeTy-Junc) whose coerce functions $[\![\mathcal{D}_1]\!]_{::}$ and $[\![\mathcal{D}_2]\!]_{::}$ are applied to the left and the right component respectively.

Similarly, rule (CoeTy-Func), which captures function calls, promotes the rewrites to its arguments. Rule (CoeTy-FuncBlock) captures the functions that accept a block as an argument. Here, coercion is applied to its first argument (the caller) and injected into the body of the block argument.

The bridge between the subtype coercion and the type coercion is provided by the function (CoeTy-Sub), which operates on the subsumption rule (Ty-Sub) of the type system. Whenever subsumption occurs somewhere in the derivation tree, associated with a type statement, the term has to be transformed in order to replace

	Expression	**Rule**
$\llbracket \langle 1 \rangle \rrbracket_{::}$	$=$ `table[Users]`	(CoeTy-Tab)
$\llbracket \langle 2 \rangle \rrbracket_{::}$	$=$ `proj(var[`x`],id)`	(CoeTy-RecProj)
$\llbracket \langle 3 \rangle \rrbracket_{::}$	$=$ `map(`$\llbracket \langle 1 \rangle \rrbracket_{::}$`,`$\lambda x.\llbracket \langle 2 \rangle \rrbracket_{::}$`)` $=$ `map(table[Users],`$\lambda x.$`proj(var[`x`],id))`	(CoeTy-FuncBlock)
$\llbracket \langle 4 \rangle \rrbracket_{::}$	$=$ `(int[23],int[42])`	(CoeTy-TupCons)
$\llbracket \langle 5 \rangle \rrbracket_{\lessdot}$	$= \lambda t.$`<`$[\cdot]^{\mathsf{T}}$`(`t`,1),`$[\cdot]^{\mathsf{T}}$`(`t`,2)>`	(CoeSubTy-Tup-List)
$\llbracket \langle 6 \rangle \rrbracket_{::}$	$= \llbracket \langle 5 \rangle \rrbracket_{\lessdot}\ \llbracket \langle 4 \rangle \rrbracket_{::}$ $=$ `<`$[\cdot]^{\mathsf{T}}$`(`$\llbracket \langle 4 \rangle \rrbracket_{::}$`,1),`$[\cdot]^{\mathsf{T}}$`(`$\llbracket \langle 4 \rangle \rrbracket_{::}$`,2)>` $=$ `<`$[\cdot]^{\mathsf{T}}$`((int[23],int[42]),1),` $[\cdot]^{\mathsf{T}}$`((int[23],int[42]),2)>`	(CoeTy-Sub)
$\llbracket \langle 7 \rangle \rrbracket_{::}$	$=$ `concat`$^{\mathsf{L}}$	(CoeTy-Sub)
$\llbracket \langle \star \rangle \rrbracket_{::}$	$=$ `concat`$^{\mathsf{L}}$`(` `map(table[Users],`$\lambda x.$`proj(var[`x`],id)),` `<`$[\cdot]^{\mathsf{T}}$`((int[23],int[42]),1),` $[\cdot]^{\mathsf{T}}$`((int[23],int[42]),2))>`	(CoeTy-Func)

Table 2.8: Coercion on the type proof tree of Query Q_1. The bullets $\langle n \rangle$ refer to the partial proof trees in Table 2.6. Based on subtyping, the tuple is automatically converted into a list. In this constellation the result of `map()` and `(int[23],int[42])` may be faithfully used as arguments of `concat`$^{\mathsf{L}}$`()`.

subtyping with runtime coercions. Function (CoeTy-Sub) expresses this intention by applying the Switch expression (in $\llbracket \mathcal{D} \rrbracket_{::}$) to the rewrite function ($\llbracket \mathcal{C} \rrbracket_{\lessdot}$) that was derived based on the derivation tree of the subtype statement ($\tau \lessdot \sigma$).

Table 2.9: Coercion over the proof tree of the type system

$$\left[\!\left[\frac{}{\Gamma \vdash \texttt{int[}i\texttt{]} :: \wr i \int}\text{(TY-INT)} \right]\!\right]_{::} = \texttt{int[}i\texttt{]} \quad \text{(COETY-INT)}$$

$$\left[\!\left[\frac{}{\Gamma \vdash \texttt{dec[}d\texttt{]} :: \mathit{Dec}}\text{(TY-DEC)} \right]\!\right]_{::} = \texttt{dec[}d\texttt{]} \quad \text{(COETY-DEC)}$$

$$\left[\!\left[\frac{}{\Gamma \vdash \texttt{str[}\mathrm{s}\texttt{]} :: \mathit{Str}}\text{(TY-STR)} \right]\!\right]_{::} = \texttt{str[}\mathrm{s}\texttt{]} \quad \text{(COETY-STR)}$$

$$\left[\!\left[\frac{}{\Gamma \vdash \texttt{bool[}b\texttt{]} :: \mathit{Bool}}\text{(TY-BOOL)} \right]\!\right]_{::} = \texttt{bool[}b\texttt{]} \quad \text{(COETY-BOOL)}$$

$$\left[\!\left[\frac{\Gamma\, x = \tau}{\Gamma \vdash \texttt{var[}x\texttt{]} :: \tau}\text{(TY-VAR)} \right]\!\right]_{::} = \texttt{var[}x\texttt{]} \quad \text{(COETY-VAR)}$$

$$\left[\!\left[\frac{}{\Gamma \vdash \texttt{()} :: ()}\text{(TY-TUPEMPTY)} \right]\!\right]_{::} = \texttt{()} \quad \text{(COETY-TUPEMPTY)}$$

$$\left[\!\left[\frac{}{\Gamma \vdash \texttt{\{\}} :: \{\}}\text{(TY-RECEMPTY)} \right]\!\right]_{::} = \texttt{\{\}} \quad \text{(COETY-RECEMPTY)}$$

$$\left[\!\left[\frac{}{\Gamma \vdash \texttt{<>} :: [\tau]} \right]\!\right]_{::} = \texttt{<>} \quad \text{(COETY-LISTEMPTY)}$$

continued on next page

$$\left[\!\!\left[\frac{\mathcal{D}_i \therefore \Gamma \vdash e_i :: \tau_i \,\Big|_{i=1,\ldots,n}}{\Gamma \vdash \texttt{(}\bar{e}\texttt{)} :: (\bar{\tau})}\text{(Ty-TupCons)}\right]\!\!\right]_{::} = \texttt{(}[\![D_1]\!]_{::}\texttt{,} \ldots \texttt{,}[\![D_n]\!]_{::}\texttt{)} \quad \text{(CoeTy-TupCons)}$$

$$\left[\!\!\left[\frac{\mathcal{D}_i \therefore \Gamma \vdash e_i :: \tau_i \,\Big|_{i=1,\ldots,n}}{\Gamma \vdash \texttt{\{}\bar{\iota} \Rightarrow \bar{e}\texttt{\}} :: \{\bar{\iota}\colon \bar{\tau}\}}\text{(Ty-RecCons)}\right]\!\!\right]_{::} = \texttt{\{}\iota_1\texttt{=>}[\![\mathcal{D}_1]\!]_{::}\texttt{,} \ldots \texttt{,}\iota_n\texttt{=>}[\![\mathcal{D}_n]\!]_{::}\texttt{\}} \quad \text{(CoeTy-RecCons)}$$

$$\left[\!\!\left[\frac{\mathcal{D}_i \therefore \Gamma \vdash e_i :: \tau \,\Big|_{i=1,\ldots,n}}{\Gamma \vdash \texttt{<}\bar{e}\texttt{>} :: [\tau]}\right]\!\!\right]_{::} = \texttt{<}[\![D_1]\!]_{::}\texttt{,} \ldots \texttt{,}[\![D_n]\!]_{::}\texttt{>} \quad \text{(CoeTy-ListCons)}$$

$$\left[\!\!\left[\frac{\text{reflection on the database schema}}{\Gamma \vdash \texttt{table[R]} :: [\{\bar{\iota}\colon \bar{\tau}\}]}\text{(Ty-TableRef)}\right]\!\!\right]_{::} = \texttt{table[R]} \quad \text{(CoeTy-Tab)}$$

$$\left[\!\!\left[\frac{\mathcal{D} \therefore \Gamma \vdash e :: \{\ldots, \iota\colon \tau, \ldots\}}{\Gamma \vdash \texttt{proj(}e\texttt{,}\iota\texttt{)} :: \tau}\text{(Ty-RecProj)}\right]\!\!\right]_{::} = \texttt{proj(}[\![\mathcal{D}]\!]_{::}\texttt{,}\iota\texttt{)} \quad \text{(CoeTy-RecProj)}$$

continued on next page

$$\left[\!\!\left[\frac{\mathcal{D}_i \therefore\ \Gamma \vdash e_i :: \tau \,\big|_{i=1,2,3}}{\Gamma \vdash \texttt{if(}e_1\texttt{,}e_2\texttt{,}e_3\texttt{)} :: \tau}\,\scriptstyle(\textsc{Ty-If}) \right]\!\!\right]_{::} = \texttt{if(}[\![\mathcal{D}_1]\!]_{::}\texttt{,}[\![\mathcal{D}_2]\!]_{::}\texttt{,}[\![\mathcal{D}_3]\!]_{::}\texttt{)} \quad (\textsc{CoeTy-If})$$

$$\left[\!\!\left[\frac{\mathrm{num}(\tau) \qquad \mathcal{D}_i \therefore\ \Gamma \vdash e_i :: \tau \,\big|_{i=1,2}}{\Gamma \vdash \circledast\texttt{(}e_1, e_2\texttt{)} :: \tau}\,\scriptstyle(\textsc{Ty-Arith}) \right]\!\!\right]_{::} = \circledast\texttt{(}[\![\mathcal{D}_1]\!]_{::}, [\![\mathcal{D}_2]\!]_{::}\texttt{)} \quad (\textsc{CoeTy-Arith})$$

$$\left[\!\!\left[\frac{\mathrm{atom}(\tau) \qquad \mathcal{D}_i \therefore\ \Gamma \vdash e_i :: \tau \,\big|_{i=1,2}}{\Gamma \vdash \olessthan\texttt{(}e_1, e_2\texttt{)} :: \mathit{Bool}}\,\scriptstyle(\textsc{Ty-Comp}) \right]\!\!\right]_{::} = \olessthan\texttt{(}[\![\mathcal{D}_1]\!]_{::}, [\![\mathcal{D}_2]\!]_{::}\texttt{)} \quad (\textsc{CoeTy-Comp})$$

$$\left[\!\!\left[\frac{\mathcal{D}_i \therefore\ \Gamma \vdash e_i :: \mathit{Bool} \,\big|_{i=1,2}}{\Gamma \vdash \owedge\texttt{(}e_1, e_2\texttt{)} :: \mathit{Bool}}\,\scriptstyle(\textsc{Ty-Junc}) \right]\!\!\right]_{::} = \owedge\texttt{(}[\![\mathcal{D}_1]\!]_{::}, [\![\mathcal{D}_2]\!]_{::}\texttt{)} \quad (\textsc{CoeTy-Junc})$$

$$\left[\!\!\left[\frac{\begin{array}{c}\mathcal{D}_m \therefore\ \Gamma \vdash m :: \tau_1 \times \ldots \times \tau_n \to \sigma \\ \mathcal{D}_i \therefore\ \Gamma \vdash e_i :: \tau_i \,\big|_{i=1,\ldots,n}\end{array}}{\Gamma \vdash m\texttt{(}\bar{e}\texttt{)} :: \sigma}\,\scriptstyle(\textsc{Ty-Func}) \right]\!\!\right]_{::} = [\![\mathcal{D}_m]\!]_{::}\texttt{(}[\![\mathcal{D}_1]\!]_{::}\texttt{,}\ \ldots\ \texttt{,}[\![\mathcal{D}_n]\!]_{::}\texttt{)} \quad (\textsc{CoeTy-Func})$$

continued on next page

$$\left\llbracket \frac{\begin{array}{c}\mathcal{D}_m \therefore \Gamma \vdash m :: [\tau] \times (\tau \to \tau_b) \to \sigma \\ \mathcal{D} \therefore \Gamma \vdash e :: \tau \quad \mathcal{D}_b \therefore \Gamma, x :: \tau \vdash e_b :: \tau_b\end{array}}{\Gamma \vdash m\texttt{(}e, \lambda x.e_b\texttt{)} :: \sigma}\,(\textsc{Ty-FuncBlock}) \right\rrbracket_{::} = \llbracket \mathcal{D}_m \rrbracket_{::}\texttt{(}\mathcal{D}\texttt{,}\,\lambda x.\mathcal{D}_b\texttt{)} \quad (\textsc{CoeTy-FuncBlock})$$

$$\left\llbracket \frac{\mathcal{D} \therefore \Gamma \vdash e :: \sigma \qquad \mathcal{C} \therefore \sigma \lessdot \tau}{\Gamma \vdash e :: \tau}\,(\textsc{Ty-Sub}) \right\rrbracket_{::} = \llbracket \mathcal{C} \rrbracket_{\lessdot} \llbracket \mathcal{D} \rrbracket_{::} \quad (\textsc{CoeTy-Sub})$$

2.9 Removing Tuple-Related Operations

The last rewrite step taking place on the expression of the language is the removal of tuple-related functions. We heavily rest on the type system to obtain the proper information about types that are necessary to perform this kind of rewrites.

We devise a rewrite function that operates on derivation trees of type system conclusions, as shown in Section 2.8.2. Similarly, we use the notation $[\![\cdot]\!]_{()}$ to express the rewrite function that, given a derivation tree $\mathcal{D}$ for a type statement $\Gamma \vdash e :: \tau$, derives a new SWITCH term in which all tuple-related functions have been replaced by means of simple positional access. The result of $[\![\mathcal{D}]\!]_{()}$ is a SWITCH term.

We must consider each type shape separately to provide a proper rewrite rule. However, in Table 2.10 we only list the rewrite rules that actually perform a rewrite. We refrain from depicting the remaining rules because they merely promote the rewrites to the constituting elements of an expression. Note that we can ignore the subtype relation ((TY-SUB) from Table 2.4), which has already been "compiled away".

In rule (REMTUP-CONCAT) we place the elements of the constituent expressions (e_1 and e_2) consecutively into a fresh tuple. The information about the number of elements present in e_1 and e_2 is delivered by the type system.

Accessing the first element and the last element is respectively captured by rule (REMTUP-FIRST) and rule (REMTUP-LAST). Whereas the first tuple component can be straightforwardly accessed, to access the last position we have to consult the type system on the number of elements in e. This number can then be used to access the last element.

Rule (REMTUP-TAKE) keeps the prefix of a tuple by tidying up the first elements of e into a fresh tuple. Rule (REMTUP-LAST) retains the suffix by consulting the type system on the number of elements present in e.

Rule (REMTUP-REVERSE) mimics the semantics of `reverse`$^{\mathsf{T}}$`()` by reverting the order of the elements manually. The length of a tuple (rule (REMTUP-LENGTH)) can be determined at compile time by having recourse to the type system.

Table 2.10: Remove tuple functions.

$$\left\llbracket \frac{\begin{array}{c}\Gamma \vdash \mathtt{concat}^{\mathsf{T}} :: (\bar\tau) \times (\bar\sigma) \to (\bar\tau, \bar\sigma) \\ \mathcal{D}_1 \therefore \Gamma \vdash e_1 :: (\bar\tau) \quad \mathcal{D}_2 \therefore \Gamma \vdash e_2 :: (\bar\sigma)\end{array}}{\Gamma \vdash \mathtt{concat}^{\mathsf{T}}\mathtt{(}e_1\mathtt{,}e_2\mathtt{)} :: (\bar\tau, \bar\sigma)} \right\rrbracket_{()} = \begin{array}{l}\mathtt{(}\mathtt{[\cdot]}^{\mathsf{T}}\mathtt{(}\llbracket\mathcal{D}_1\rrbracket_{()}\mathtt{,}1\mathtt{)}\mathtt{,}\dots\mathtt{,}\mathtt{[\cdot]}^{\mathsf{T}}\mathtt{(}\llbracket\mathcal{D}_1\rrbracket_{()}\mathtt{,}n\mathtt{)}\mathtt{,} \\ \quad \mathtt{[\cdot]}^{\mathsf{T}}\mathtt{(}\llbracket\mathcal{D}_2\rrbracket_{()}\mathtt{,}1\mathtt{)}\mathtt{,}\dots\mathtt{,}\mathtt{[\cdot]}^{\mathsf{T}}\mathtt{(}\llbracket\mathcal{D}_2\rrbracket_{()}\mathtt{,}m\mathtt{)}\ \mathtt{)}\end{array} \qquad \text{(RemTup-Concat)}$$

$$\left\llbracket \frac{\Gamma \vdash \mathtt{first}^{\mathsf{T}} :: (\bar\tau) \to \tau_1 \qquad \mathcal{D} \therefore \Gamma \vdash e :: (\bar\tau)}{\Gamma \vdash \mathtt{first}^{\mathsf{T}}\mathtt{(}e\mathtt{)} :: \tau_1} \right\rrbracket_{()} = \mathtt{[\cdot]}^{\mathsf{T}}\mathtt{(}\llbracket\mathcal{D}\rrbracket_{()}\mathtt{,}1\mathtt{)} \qquad \text{(RemTup-First)}$$

$$\left\llbracket \frac{\Gamma \vdash \mathtt{last}^{\mathsf{T}} :: (\bar\tau) \to \tau_n \qquad \mathcal{D} \therefore \Gamma \vdash e :: (\bar\tau)}{\Gamma \vdash \mathtt{last}^{\mathsf{T}}\mathtt{(}e\mathtt{)} :: \tau_n} \right\rrbracket_{()} = \mathtt{[\cdot]}^{\mathsf{T}}\mathtt{(}\llbracket\mathcal{D}\rrbracket_{()}\mathtt{,}n\mathtt{)} \qquad \text{(RemTup-Last)}$$

$$\left\llbracket \frac{\begin{array}{c}\Gamma \vdash \mathtt{take}^{\mathsf{T}} :: (\bar\tau) \times \wr i \int \to (\tau_1, \dots, \tau_i) \\ \mathcal{D} \therefore \Gamma \vdash e :: (\bar\tau)\end{array}}{\Gamma \vdash \mathtt{take}^{\mathsf{T}}\mathtt{(}e\mathtt{,int[}i\mathtt{])} :: (\tau_1, \dots, \tau_i)} \right\rrbracket_{()} = \mathtt{(}\mathtt{[\cdot]}^{\mathsf{T}}\mathtt{(}\llbracket\mathcal{D}\rrbracket_{()}\mathtt{,}1\mathtt{)}, \dots, \mathtt{[\cdot]}^{\mathsf{T}}\mathtt{(}\llbracket\mathcal{D}\rrbracket_{()}\mathtt{,}i\mathtt{))} \qquad \text{(RemTup-Take)}$$

continued on next page

$$\left\llbracket \frac{\begin{array}{c}\Gamma \vdash \texttt{drop}^{\mathsf{T}} :: (\bar\tau) \times \lbag i \rbag \to (\tau_{i+1},\ldots,\tau_n) \\ \mathcal{D} \therefore \Gamma \vdash e :: (\bar\tau)\end{array}}{\Gamma \vdash \texttt{drop}^{\mathsf{T}}\texttt{(}e\texttt{,int[}i\texttt{])} :: (\tau_{i+1},\ldots,\tau_n)} \right\rrbracket_{()} = \texttt{([}\cdot\texttt{]}^{\mathsf{T}}\texttt{(}\llbracket\mathcal{D}\rrbracket_{()}\texttt{,}i+1\texttt{)},\ldots,\texttt{[}\cdot\texttt{]}^{\mathsf{T}}\texttt{(}\llbracket\mathcal{D}\rrbracket_{()}\texttt{,}n\texttt{))} \quad \text{(RemTup-Drop)}$$

$$\left\llbracket \frac{\begin{array}{c}\Gamma \vdash \texttt{reverse}^{\mathsf{T}} :: (\tau_1,\ldots,\tau_n) \to (\tau_n,\ldots,\tau_1) \\ \mathcal{D} \therefore \Gamma \vdash e :: (\tau_1,\ldots,\tau_n)\end{array}}{\Gamma \vdash \texttt{reverse}^{\mathsf{T}}\texttt{(}e\texttt{)} :: (\tau_n,\ldots,\tau_1)} \right\rrbracket_{()} = \texttt{([}\cdot\texttt{]}^{\mathsf{T}}\texttt{(}\llbracket\mathcal{D}\rrbracket_{()}\texttt{,}n\texttt{)},\ldots,\texttt{[}\cdot\texttt{]}^{\mathsf{T}}\texttt{(}\llbracket\mathcal{D}\rrbracket_{()}\texttt{,}1\texttt{))} \quad \text{(RemTup-Reverse)}$$

$$\left\llbracket \frac{\begin{array}{c}\Gamma \vdash \texttt{length}^{\mathsf{T}} :: (\bar\tau) \to \lbag n \rbag \\ \mathcal{D} \therefore \Gamma \vdash e :: (\bar\tau)\end{array}}{\Gamma \vdash \texttt{length}^{\mathsf{T}}\texttt{(}e\texttt{)} :: \lbag n \rbag} \right\rrbracket_{()} = \texttt{int[}n\texttt{]} \quad \text{(RemTup-Length)}$$

2.10 Related Work

Beside the embedding of query languages, the potential of Ruby as a host language for a diversity of DSL's has been already recognized. For example, in [AC+11a] Alvaro, Conway, et al. endow Ruby with a DSL called Bloom, which facilitates *distributed programming* on key-value stores. Bloom programs are bundles of declarative statements about a collection of tuples, similar to SQL views. The authors identify a set of Ruby methods—such as `map()`, `reduce()`, and `group()`—so that an implementation on top of a distributed programming environment remains feasible.

In a different setting, Furr, An, et al. present a static type inference that blends Ruby's type system with a static typing discipline [FA+09]. In this way, the benefits of combined dynamic and static typing can be fully exploited. They describe a type language incorporating various concepts—including union and intersection types [Pie91], object types [CM96], and parametric polymorphism [Pie02]—to closely resemble Ruby's dynamic type system. Like in this thesis, the authors use intersection types to track different behavior of Ruby methods depending on the types of their parameters.

A different approach has been proposed in [AC+11b] by An, Chaudhuri, et al. The authors present a technique based on dynamic program executions to infer static types. This *constraint-based dynamic type inference* wraps each run-time value to connect it with a type variable and generated constraints that must hold when the value is used.

In a subsequent paper, An, Chaudhuri, and Foster expose the semantics of query methods used in Ruby on Rails [ACF09] by rewriting them into pure Ruby code. The authors then leverage the techniques described in the previous chapter to track errors made in Rails at compile time.

Chapter 3

A Relational Portrayal of Switch

In the current chapter we aim to trade the iterative semantics of Switch for the efficient set-oriented execution model of a database. In the course of this, we rely on a compilation scheme, called *loop lifting*, which originated as a technique to express XQuery's side effect-free FLWOR-expressions by relational means [see GST04; Teu06].

In the course of the last years this technique has been leveraged to full support of languages (or subsets of languages) whose semantic foundation is based on *list comprehensions* [TP89; Wad92]. List comprehensions revolve around the iterative side-effect-free evaluation of expressions under the binding of an unmodifiable iteration variable. They have been widely recognized as a concise and neat notation to perform complex computations over lists and feature in a substantial amount of the programming languages in vogue today—among them Haskell [Has], Java [Jav] and Python [Pyt] to name a few.

With the **`Enumerable`** mixin we already identified an iteration-centric subset of Ruby based on an iterative core represented by the method **`each`**. This method, whose sole purpose is to expose all elements of a collection in a single iteration, lays the foundation for the entire functionality provided by the module. The Switch snippet below, for instance, denotes the list resulting from a derivation of the body e_{body} independently for each element of e—a constellation that perfectly blends with loop lifting:

$$e\texttt{.map \{|}x\texttt{|}\ e_{\texttt{body}}\texttt{\}}$$

Whereas vanilla Ruby permits side-effecting computations—such as variable assignments—in e_{body}, Switch proves to be sufficiently shielded:

(i) Whenever Ruby's runtime encounters a Switch expression it starts to arrange the corresponding expression tree.

(ii) In the course of this process, any expression eluding the control of SWITCH is subject to RUBY's runtime engine and directly reduced to a constant value that may be seamlessly integrated into the expression tree.

The execution of the resulting SWITCH expression is deferred until the elements of the computation are actually consumed in the host language. Even an assignment to a variable named x in the loop body of the above expression will only shadow the iteration variable and thus not influence the behavior of the loop. The lack of side effects is an essential requirement for a language to qualify for a loop-lifted translation: because the iterations cannot interfere, they may be evaluated in arbitrary order, which perfectly suits the set-oriented execution model of relational database systems.

In the next section we provide a bird's-eye view of loop lifting and draw our attention on how SWITCH values are represented in a tabular fashion in Section 3.2. In Section 3.3 we shed light on the tabular operators that constitute a target language for the compilation rules in the remaining chapter. A thorough description of loop lifting can be found in [Teu06].

3.1 A Primer in Loop Lifting

The primary concern of loop lifting is the sound implementation of iteration constructs by means of a relational representation. Any compiler that employs loop lifting to translate loop-centric programs (or queries) into a database-executable form emits a plan or even several plans comprised of relational operators.

To illustrate this, consider the following RUBY snippet[1] in which elements e_1 to e_n are successively bound to the variable x, which in turn is possibly accessed in e_{body} to evaluate its constituents that jointly form the overall expression result:

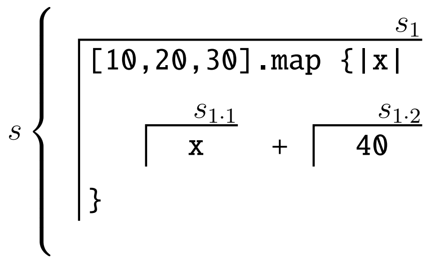
$$\begin{gathered}\texttt{[}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{].map \{|}x\texttt{| } e_{\textsf{body}}\texttt{\}} \\ \equiv \\ \texttt{[}e_{\textsf{body}}\,[x \mapsto e_1]\texttt{,}\ldots\texttt{,}e_{\textsf{body}}\,[x \mapsto e_n]\texttt{]}\end{gathered}$$

(The expression $e\,[x \mapsto e_i]$ denotes the replacement of all free occurrences of x in e by e_i.) Since the semantics of `map` is purely functional (the evaluation of the body e_{body} is performed independently for each element e_1 to e_n) e_{body} may be evaluated in arbitrary order, or even in parallel.

The intuition behind loop lifting is the avoidance of an explicit iterator construct, such as `map`, by "unrolling" an iteration by means of a *loop* table with a single column `iter`; n iterations lead to a *loop* table containing exactly n values $(1, \ldots, n)$.

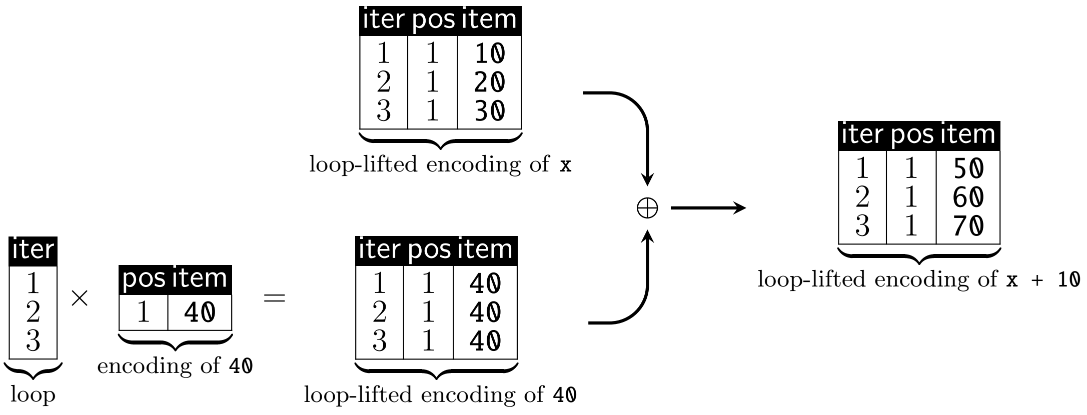
[1]Note that this expression is only valid in SWITCH if the literal RUBY array, which is represented as a tuple, may be regarded as a list, *i.e.*, all elements evaluate to a common type.

Iteration constructs in Switch may be arbitrarily nested: each iteration construct establishes a new iteration context in which nested expressions are evaluated (in general, this leads to a tree shaped hierarchy of iteration scopes). Any expression in Switch is considered to be iteratively evaluated in scope $s_{x \cdot y}$ of its directly enclosing iteration scope s_x; the top level expression is evaluated in a pseudo single-iteration denoted by s. The relational representation of a subexpression within an iteration scope, such as the expression `x + 40` within scope s_1



```
[10,20,30].map {|x|
    x  +  40
}
```

has to be *lifted* with respect to the *loop* table as shown in figure Figure 3.1.



iter	pos	item
1	1	10
2	1	20
3	1	30

loop-lifted encoding of `x`

iter
1
2
3

loop

pos	item
1	40

encoding of `40`

iter	pos	item
1	1	40
2	1	40
3	1	40

loop-lifted encoding of `40`

iter	pos	item
1	1	50
2	1	60
3	1	70

loop-lifted encoding of `x + 10`

Figure 3.1: Loop-lifted representation of x `+ 10` within scope s_1.

In the following, we use $\overline{\mathsf{item}}$ as a shortcut for $\mathsf{item}_1, \ldots, \mathsf{item}_\mathsf{n}$, and likewise $\bar{v}$ to denote the sequence $v_1, \ldots, v_n$. The table schema $\langle \mathsf{iter}, \mathsf{pos}, \overline{\mathsf{item}} \rangle$ is used pervasively in the loop lifting compilation. For an expression e, a single loop-lifted row encoding $\langle i, p, \bar{v} \rangle$ may be read as "In iteration i the expression e yields the values v_1 to v_n at the list position that corresponds to the rank of p in column pos." Establishing a unified table schema is crucial to ensuring the compositionality of the translation rules we give in Section 3.4. Following the description of how values of Switch are represented in a table, we will take a look at relational algebra that forms the intermediate language in our compilation scheme.

3.2 A Relational Representation of Values

Each value in SWITCH has its counterpart in the relational realm. In SWITCH a literal of the base type is by definition a value. Decomposable data types, such as tuples and lists, are values if their constituents are values as well. Values, as stated above, play an important role because they describe the result of a program execution in SWITCH; the type system makes sure that only programs resulting in values (listed below) qualify as valid SWITCH programs which may translated into a database executable form. In the following list we consider each type in SWITCH and describe how associated values are encoded in a tabular fashion.

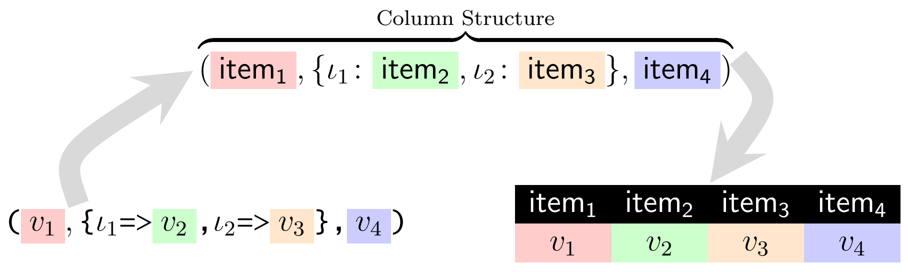
Base types (*Int*, *Str*, . . .). A literal value ℓ that is typed as atomic base type is represented as a table with a single column item_1 and a single row containing the value to represent:

item_1
ℓ

.

Tuples and Records. The encoding scheme for tuples and records is very similar to that of the base types: Each of the constituents of either a tuple or a record inhabits a different column in its tabular representation. For the values of both types we adopt the "item_1 to item_n"-naming scheme for columns.

Figure 3.2 illustrates how the tuple value $(v_1, \{\iota_1 \texttt{=>} v_2, \iota_2 \texttt{=>} v_3\}, v_4)$ is encoded in a table; v_1 to v_4 are values of a base type. Even though the tuple is nested—another decomposable value in form of a record with labels ι_1 and ι_2 is placed on the second position—its tabular representation does not exceed a single row. The column structure maps each of the atomic values to its column and retains the shape of the result type.


Column Structure
$(\mathsf{item}_1, \{\iota_1: \mathsf{item}_2, \iota_2: \mathsf{item}_3\}, \mathsf{item}_4)$
$(v_1, \{\iota_1 \texttt{=>} v_2, \iota_2 \texttt{=>} v_3\}, v_4)$

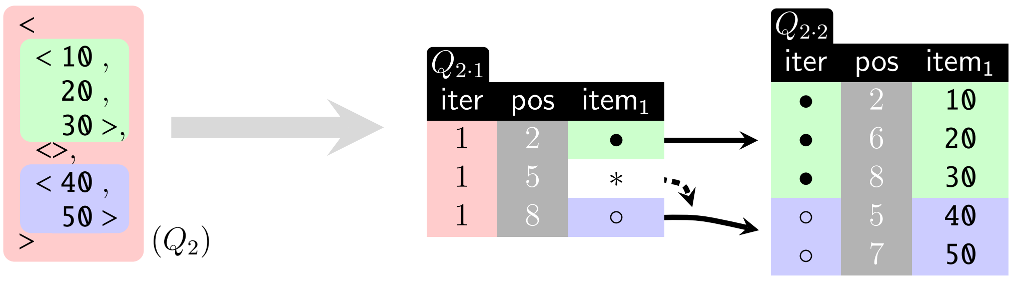
item_1	item_2	item_3	item_4
v_1	v_2	v_3	v_4

Figure 3.2: The representation of a nested tuple value in a tabular fashion. The column structure provides information on how the individual components of a decomposable type are flattened to the table schema.

Lists. Ordered lists present the principal aggregate data structure in SWITCH. Because our intermediate language in form of a table algebra operates over inherently unordered, tables we encode order into the tabular representation by means of a pos column that properly reflects the logical order of the elements within a list. It is critical to encode positions in the data itself to ensure a communication from statement to statement (which would be impossible by SQL's `ORDER BY` construct). The pos column may host values of an arbitrary domain as long as a linear order may be imposed on them. This observation offers some serious optimization potential as described in [Rit10]. In this work however we will populate the pos column with natural numbers only. The scheme by which the elements $<v_1, \ldots, v_m>$—with v_1 to v_m being values of an arbitrary but equal type—are arranged in a table is shown on the right. Note that we used $\overline{\text{item}}$ since we do not know how many columns these values occupy in their tabular representation.

pos	$\overline{\text{item}}$
1	$\overline{v_1}$
$\vdots$	$\vdots$
m	$\overline{v_m}$

Since relations adhere to the first normal form—the domain of an attribute only contains atomic values[2]—an arbitrary nested list cannot be hosted by a single table. Lists of nesting depth n are thus distributed over exactly n tables T_1 to T_n, each of which frames one nesting level. Two tables T_i and T_{i+1} are connected by a foreign-key relationship based on surrogates values.



iter	pos	item$_1$
1	2	•
1	5	∗
1	8	∘

iter	pos	item$_1$
•	2	10
•	6	20
•	8	30
∘	5	40
∘	7	50

Figure 3.3: Representation of a nested lists by relational means. Between the Outer table and the Inner table a foreign-key relationship is established to properly reflect the shape of the list.

For further illustration consider the nested list and its corresponding tabular representation depicted in Figure 3.3. The outer shape of the list is captured by the table Outer: each of the surrogate values •, ∗ and ∘ in the column item$_1$ stands for an entire list located in the table Inner. A list in Inner is constituted

[2]The demand for atomic values in relational database systems has been decreased with the advent of the SQL:1999 standard by which the composite types array and anonymous row have been introduced [MS01].

by the group of rows that contain equal iter values; the column iter assumes the role of a list identifier. Note that the value $*$ stands for the empty list and thus does not find a matching value in the iter column. For the "glueing" columns (item_1 and iter) in table $Q_{2\cdot 1}$ and $Q_{2\cdot 2}$, values of any domain are suitable as long as they support some notion of equivalence.

3.3 A Relational Algebra for SWITCH

The table operators lined up in Table 3.1 resemble a standard or restricted form of operators found in the classical relational algebra [Cod70]—each of these operators considers one or more tables as input and produces another table as a result. While these operators are agnostic about an actual database system, they reflect the query capabilities of any modern database system at the same time. Each operator is, by design, furnished with simple semantics: for the select operator σ_a, *e.g.*, it is sufficient to only select tuples with $a = \texttt{true}$. Similarly the equi-join operator ($\overset{=}{\bowtie}_{a=b}$) only supports equality predicates as join condition. The advantage of these RISC-style operators is that they are especially beneficial to efficient algebraic optimization and rewrites as proposed in [GMR09; Rit10; Teu06].

Operator	**Description**
$\pi_{a_1:b_1,\dots,a_n:b_n}(\cdot)$	projection and column renaming from a_i into b_i
$\sigma_a(\cdot)$	select rows by $a = \texttt{true}$
$@_{a:v}(\cdot)$	add column a with values v
$\delta(\cdot)$	duplicate elimination
$\varrho_{a:\langle b_1,\dots,b_n\rangle \mid p}(\cdot)$	row numbering (with order $\langle b_1,\dots,b_n\rangle$ and partition by p)
$\vec{\Cup}_{a:\langle b_1,\dots,b_n\rangle}(\cdot)$	row ranking (with order $\langle b_1,\dots,b_n\rangle$)
$\circledcirc_{a:\langle b_1,b_2\rangle}(\cdot)$	2-ary arithmetic/comparison/boolean operator
$\cdot \overset{=}{\bowtie}_{a=b} \cdot,$	equi-join
$\cdot \times \cdot$	cross product
$\cdot \uplus \cdot, \cdot \setminus \cdot$	disjoint union and difference
$\mathrm{GRP}_{a:\circ(b)/g}(\cdot)$	aggregation on b group by g ($\circ \in \{\textsc{max}, \textsc{sum}, \textsc{count}, \dots\}$)
$\boxed{\begin{matrix} a_1 \dots a_n \\ \emptyset \end{matrix}}, \boxed{\begin{matrix} a_1 \dots a_n \\ \square\square\square \end{matrix}}$	(empty) literal table with columns $a_1, \dots, a_n$
$\mathsf{T}_{(a_1,\dots,a_n)\langle p\rangle}$	database table reference to T with columns $a_1, \dots, a_n$ and primary key p
$♀_{a_1,a_2,\langle a_3,\dots,a_n\rangle}(\cdot)$	plan root (with payload columns $\langle a_3, \dots, a_n\rangle$

Table 3.1: *Assembly style* table operators reflect the query capabilities of modern databases systems. Their simple semantics is advantageous when it comes to effective algebraic optimization.

Among the few non-standard operators is the attach operator $@_{a_1:v_1,\dots,a_n:v_n}(q)$, which is a shortcut for $\begin{array}{|c|c|c|}\hline a_1 & \dots & a_n \\ \hline v_1 & \dots & v_n \\ \hline \end{array} \times q$. Both the row number and the row rank operator are a tribute to arbitrary nesting and order in Switch. The pos column, which is used to reflect order, is populated with numeric values adhering to an order criterion that is imposed by the parameters of these operators.

Given an order $b_1, \dots, b_n$, the row-number operator $\varrho_{a:\langle b_1,\dots,b_n\rangle|p}(q)$ generates consecutive row numbers in the new column a for every partition p in q. In each partition, row numbering re-starts at 1. Omitting parition, we write $\varrho_{a:\langle b_1,\dots,b_n\rangle}(q)$ to express that we operate on a single partition that spans the entire table embodied by expression q.

In Switch, the other variant of numbering in form of the rank operator is usually considered to maintain order. The row rank $\vec{\Cup}_{a:\langle b_1,\dots,b_n\rangle}(q)$ populates the column a with the rank of the "current" row among the other rows in the table q. If rows are tied—*i.e.* they have equal values in the ordering column—the same rank will be assigned to them. In the compilation scheme, the row rank operator meets various duties that are directly or implicitly linked with maintaining order, such as (i) generating iteration identifiers, (ii) group identifiers, (iii) absolute positions, (iv) and surrogate values to represent nested lists. On relational database systems a the rank and the row-number operator causes a considerable cost: using this operator entails a sorting operation on the input table according to the order criterion. In Section 5.1 we will briefly discuss an optimization technique that could remove these operators from the query plans in certain situations.

The proper functions for the above operators are at the ready in any SQL:1999 compliant database system as part of the SQL/OLAP amendment. In the SQL translation of the algebraic operators above (4), we translate the row number operator $\varrho_{a:\langle b_1,\dots,b_n\rangle|p}(q)$ into the following SQL-query:

```
SELECT *, ROW_NUMBER() OVER (
            PARTITION BY p ORDER BY b1,...,bn) AS a
  FROM q
```

Similarly, the row rank operator $\vec{\Cup}_{a:\langle b_1,\dots,b_n\rangle}(q)$ follows the semantics of the following SQL-query:

```
SELECT *, DENSE_RANK() OVER (ORDER BY b1,...,bn) AS a
  FROM q
```

The serialization operator $\female_{a_1,a_2,\langle a_3,\dots,a_n\rangle}(q)$ forms the plan root of every query plan. Applied on q, this primitive will divide the result into groups (or lists) based on the column a_1. The column a_2 describes the sequence order while the sequence $\langle a_3, \dots, a_n\rangle$ accomodates the payload information.

Plan Bundle

$$pb \;:=\; (\underbrace{q}_{\text{Query Plan}}, \underbrace{cs}_{\text{Column Structure}}, \underbrace{surr}_{\text{Surrogate Map}})$$

Column Structure

$$\begin{array}{rcll} cs & := & \mathsf{item_n} & \text{Columns} \\ & | & (cs, \ldots, cs) & \text{Tuples} \\ & | & \{\iota\colon cs, \ldots, \iota\colon cs\} & \text{Records} \end{array}$$

Surrogate Map

$$surr \;:=\; \{\ldots, \mathsf{item_n} \mapsto pb, \ldots\}$$

Table 3.2: Auxiliary structures used in the compilation scheme: q stands for a relational plan, the column structure is used to “flatten” the the type of a subexpression to a table schema, and the surrogate map hosts further plan bundles to properly represent nesting.

3.4 Introducing the Compilation Scheme

The inference rules featured in this chapter present the first step in our translation of a SWITCH expression (or query) into a database executable form. The principal elements in this process are compilation rules with the general shape

$$\Gamma; loop \vdash e \Rrightarrow pb \;,$$

which may be read as “A SWITCH subexpression e is compiled to its plan bundle pb,” provided that

(i) Γ denotes the variable environment that maps any variable x that occurs to be free in e to the plan bundle pb_x and
(ii) a relation $loop$ that describes the iteration context of an expression e.

The *plan bundle* comprises the essential components to render the subexpression e into its loop-lifted encoding and to properly restore the result in the host language. As we illustrate in Table 3.2, the plan bundle pb spans three parts:

(i) The query q describes the loop-lifted encoding of e in terms of table operators. It emits a table with the schema $\langle \mathsf{iter}, \mathsf{pos}, \overline{\mathsf{item}} \rangle$; the item sequence $\overline{\mathsf{item}}$ always starts at 1 and is consecutively numbered and solely parametrized by a parameter n that reflects the number of items. The translation rules operate exclusively on tables adhering to this schema. In the following, we use $\mathbb{L}^n$ to denote the type of a table with a loop-lifting schema comprising n item columns.

(ii) The items in the *column structure* reflect the types of the type system with the exception that base types are substituted by column names $\mathsf{item_n}$. Being assembled during the compilation process, the column structure properly reflects the type of e. The corresponding base types are mapped to columns in the query q, where associated values can be found. For example, the type $(\mathit{Int}, \{\iota_1\colon \mathit{Str}, \iota_2\colon \mathit{Dec}\})$ corresponds to the column structure $(\mathsf{item_1}, \{\iota_1\colon \mathsf{item_2}, \iota_2\colon \mathsf{item_3}\})$, which assigns the atomic types *Int*, *Str*, and *Dec* to the column names $\mathsf{item_1}$, $\mathsf{item_2}$, and $\mathsf{item_3}$ respectively.

(iii) If the Switch expression e is subject to nesting, this is indicated by one or more columns containing surrogate values. The column names are found in the *surrogate map* that maps column names to another plan bundle that describes the inner shape of the expression e. More accurately, the surrogate map is a partial function to which a column name may be applied. In return, we obtain the associated plan bundle.

We write $\mathit{surr} \uplus \{\mathsf{item_1} \mapsto \mathit{pb}_1, \ldots, \mathsf{item_n} \mapsto \mathit{pb}_n\}$ in order to append $\mathsf{item_1}$ to $\mathsf{item_n}$ with plan bundle pb_1 to pb_n to the surrogate map surr, and $\mathrm{dom}(\mathit{surr})$ to obtain the domain of the surrogate map. To restrict a surrogate map surr to a subset of its domain $I \subseteq \mathrm{dom}(\mathit{surr})$ we write $\mathit{surr}\,|_I$.

The compilation process works in a bottom-up manner, starting with the singleton *loop* table $\begin{array}{|c|}\hline \mathsf{iter}\\\hline 1\\\hline\end{array}$ that embodies the aforementioned pseudo single-iteration of the top-level scope s_0. The compilation rules are truly compositional, *i.e.*, the direct constituents of each expression are translated independently from their surrounding expression.

We start with the translation of Switch expressions where nesting is of secondary importance. These translation rules address an expression e by either resulting in an empty surrogate map ($\emptyset$) or by adopting the column structure derived by its direct constituents. We introduce nesting in Section 3.10 and discuss expressions that call fora careful treatment of their nested structures in the following sections.

For the sake of clarity, we use abstract syntax rather than concrete syntax in all following examples. In this way, we can safely distinguish between lists and tuples, which are perceived as arrays in Ruby syntax. However, we omit the abstract notation for literals. In a few examples, we also provide the concrete syntax to emphasize the connection to the Ruby code.

3.5 Auxiliary Functions

In the current section we devise several meta functions that facilitate the definition of the compilation rules. Each function performs a particular compile time operation, ranging from the retrieval of properties of plan components to the formulation of query-plan fragments.

3.5.1 Gathering Items

An operation that commonly occurs in the formulation of translation rules is the retrieval of columns names that are used to represent a value in a table. The function

$$\text{items}(cs)$$

is inductively defined over column structures and returns the set of item columns that are used to accommodate associated values.

This function is overloaded to work on surrogate maps in order to obtain the columns that contain surrogate values in order to express nested data structures. In mathematical terms: items($surr$) derives the input domain of the partial function $surr$. Both operations are defined in Table 3.3.

To obtain the number of item columns in a column structure we use the shortcut $|cs|$ to mean $|\text{items}(cs)|$ (set cardinality), and likewise for surrogate maps we write $|surr|$ to mean $|\text{items}(surr)|$.

$$\begin{aligned}\text{items}(\mathsf{item_n}) &= \{\mathsf{item_n}\}\\ \text{items}((cs_1,\ldots,cs_n)) &= \text{items}(cs_1) \uplus \ldots \uplus \text{items}(cs_n)\\ \text{items}(\{\iota_1\colon cs_1,\ldots,\iota_n\colon cs_n\}) &= \text{items}(cs_1) \uplus \ldots \uplus \text{items}(cs_n)\end{aligned} \qquad \text{(CS-Items)}$$

$$\text{items}(surr) = \text{dom}(surr) \qquad \text{(SM-Items)}$$

Table 3.3: Gathering the item columns that used to represent the values associated with a column structure. Applied on surrogate maps, items($surr$) obtains all columns that contain surrogate values.

3.5.2 Adjustment of Items

In the compilation rules, we heavily rely on the loop-lifted table schema comprising a consecutively numbered item sequence. When working with multiple tables, columns need to be re-adjusted to restore the loop-lifted table schema. The function

$$cs \rhd l$$

descends into the column structure and increments each $\mathsf{item_n}$ by a number l resulting in $\mathsf{item_{n+l}}$. The shape of the column structure, however, remains untouched. We use $cs \lhd l$ as a shortcut for $cs \lhd l \equiv cs \rhd (-l)$.

This function is overloaded to work on surrogate maps in which the following equivalence holds:

$$surr\ \mathsf{item_n} = (surr \rhd l)\ \mathsf{item_{n+l}} \quad .$$

Similarly, the expression $surr \lhd l$ is used to mean $surr \rhd (-l)$. Consider Table 3.4 for a complete definition of both operations.

$$\begin{aligned} \mathsf{item_n} \rhd l &= \mathsf{item_{n+l}} \\ (cs_1, \ldots, cs_n) \rhd l &= (cs_1 \rhd l, \ldots, cs_n \rhd l) \\ \{\iota_1 \colon cs_1, \ldots, \iota_n \colon cs_n\} \rhd l &= \{\iota_1 \colon cs_1 \rhd l, \ldots, \iota_n \colon cs_n \rhd l\} \end{aligned} \qquad \text{(CS-Items-Inc)}$$

$$surr \rhd l = \{\mathsf{item_{i+l}} \mapsto surr\ \mathsf{item_n} \mid \mathsf{item_i} \in \mathrm{dom}(surr)\} \qquad \text{(SM-Items-Inc)}$$

Table 3.4: Adjustment of column names in column structures and surrogate maps.

3.5.3 Lifting the Environment

The lift operator in Table 3.5 lifts each loop-lifted representation of variables in the environment Γ into a new iteration scope described by the query *map*, as it can be seen, *e.g.* in rule (LL-Map) on page 93. The query *map* relates the iteration identifiers from outer scope s_x to the inner scope $s_{x \cdot y}$ and has the schema $\langle \mathsf{outer}, \mathsf{inner}, \mathsf{pos_{outer}} \rangle$; the column $\mathsf{pos_{outer}}$ additionally preserves the logical order of the elements in the outer scope to avoid a costly re-calculation. This connection between directly enclosing scopes permits us to establish the representation of any free variable captured by the environment Γ in scope $s_{x \cdot y}$ by means of an equi-join.

3.5.4 Restricting the Environment

The restrict operator in Table 3.6 confines each loop-lifted representation of variables in the environment Γ to values described by the iteration scope in the query *loop*. The operation is used in preparation for the translation of conditionals (rule (LL-If) on page 86) to respectively assemble separate environments for the *then*-branch and the *else*-branch. The meta operation $\llbracket \cdot \rrbracket_q^{\curlywedge}$ (see Section 3.11.2 for the definition) removes the values that are subject to the confinement from (possibly) nested lists hosted by the surrogate maps.

$$\text{lift}(map, q^{l}) = \begin{array}{c} \pi_{\mathsf{inner:iter,pos,items}(cs)} \\ | \\ \bar{\bar{\bowtie}}_{\mathsf{outer=iter}} \\ / \quad \backslash \\ map \quad q \end{array} \tag{Lift}$$

$$(\!|\Gamma|\!)^{\uparrow}_{map} = \left\{ \begin{array}{l} \textbf{let } (q, cs, surr) = \Gamma\, x \\ \textbf{ in } x \mapsto (\text{lift}(map, q^{|cs|}), cs, surr) \end{array} \,\middle|\, x \in dom(\Gamma) \right\} \tag{Env-Lift}$$

Table 3.5: The operation $(\!|\Gamma|\!)^{\uparrow}_{map}$ lifts all variables in Γ into a new iteration scope. The query *map* relates two directly enclosing iteration scopes by preserving the originating outer iteration for each iteration in the inner scope. Note that the query q in Eq. (Lift) is annotated with the number of item columns (l) that appear in the schema of q.

$$\text{restrict}(loop, q^{l}) = \begin{array}{c} \pi_{\mathsf{iter,pos,item_1,\ldots,item_l}} \\ | \\ \bar{\bar{\bowtie}}_{\mathsf{iter=iter_2}} \\ / \quad \backslash \\ q \quad \begin{array}{c} \pi_{\mathsf{iter:iter_2}} \\ | \\ loop \end{array} \end{array} \tag{Restrict}$$

$$(\!|\Gamma|\!)^{\curlywedge}_{loop} = \left\{ \begin{array}{l} \textbf{let } (q, cs, surr) = \Gamma\, x\,\textbf{,} \\ \qquad\qquad\qquad q' = \text{restrict}(loop, q^{|cs|}) \\ \textbf{ in } x \mapsto (q', cs, \llbracket surr \rrbracket^{\curlywedge}_{q'}) \end{array} \,\middle|\, x \in dom(\Gamma) \right\} \tag{Env-Restrict}$$

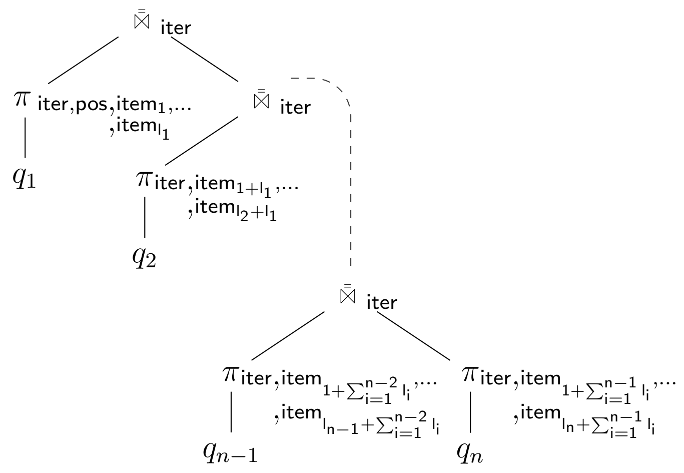
Table 3.6: The operation $(\!|\Gamma|\!)^{\curlywedge}_{loop}$ restricts all variables in Γ to the iteration context, described by the query *loop*. Observe that the query q in Eq. (Restrict) is annotated with the number of item columns (l) that appear in the schema of q.

3.5.5 Relational Zip

The relational zip operator in Table 3.7 merges the loop-lifted representation of an arbitrary but finite number of queries into a single table. It is primarily used when expressions displayed by separate queries are rendered into a single plan, such as in rule (LL-Tup) on page 72.

The queries are paired according to their iteration identifier, so that elements originating in the same iteration are placed in a single row. To prevent name clashes, the item sequences are re-adjusted accordingly. For this purpose, every query q_i is annotated with the number of columns (l_i) that are required to accommodate the intermediate result in its tabular encoding.

$$\curlyvee\,(q_1^{l_1}, q_2^{l_2}, \ldots, q_{n-1}^{l_{n-1}}, q_n^{l_n}) =$$


$\bar{\bowtie}$ iter
π iter,pos,item$_1$,... ,item$_{l_1}$
q_1
$\bar{\bowtie}$ iter
π iter,item$_{1+l_1}$,... ,item$_{l_2+l_1}$
q_2
$\bar{\bowtie}$ iter
π iter,item$_{1+\sum_{i=1}^{n-2} l_i}$,... ,item$_{l_{n-1}+\sum_{i=1}^{n-2} l_i}$
q_{n-1}
π iter,item$_{1+\sum_{i=1}^{n-1} l_i}$,... ,item$_{l_n+\sum_{i=1}^{n-1} l_i}$
q_n

(RelZip)

Table 3.7: The relational zip operator pairs the tuples of n queries originating in the same iteration. Items are automatically adjusted to prevent name clashes. For the sake of readability, we write $q_1 \mathbin{\bar{\bowtie}}_{\mathsf{iter}}\, q_2$ to mean $\pi_{\mathsf{iter,pos},\overline{\mathsf{item}}_{1\cdot 2}}(q_1 \mathbin{\bar{\bowtie}}_{\mathsf{iter=iter_2}}\, \pi_{\mathsf{iter:iter_2},\overline{\mathsf{item}}_2}(q_2))$. The item sequences $\overline{\mathsf{item}}_2$ and $\overline{\mathsf{item}}_{1\cdot 2}$ respectively denote the item columns participating in q_2 and the join result.

3.5.6 Relational Append

The relational append operator in Table 3.8 combines an arbitrary number of queries into a single query. It is used whenever the semantics of an expression includes an append operation—such as the list constructor (see on page 87).

In contrast to the relational zip operator in the preceding section, the relational append operator places the item columns that originate in the same iteration one below each other rather than in the same row. Essentially, the append operation is implemented via a disjoint union. To ensure the logical ordering— *e.g.* elements from query q_1 need to appear before elements of query q_2—we extend the queries with a temporary column (ord). Based on the schema ⟨iter, ord, pos⟩, a new position (pos_2) is established. It is important to see that the column structure (*cs*) and a surrogate map (*surr*) participate in the preparation of the query in order to obtain the item columns that are used in the queries q_1 through q_n. Note that the column pos_2 doubles in the preparation of new surrogate values for the columns in items(*surr*). The temporary column ord is further used to cope with nesting (see Section 3.11.1 on page 83).

$$\uplus^{(cs,surr)}(q_1,\ldots,q_{n-1},q_n) = \begin{tikzcd}
& \pi_{\substack{\mathsf{iter},\mathsf{pos}_2:\mathsf{pos},\mathsf{ord},\\ \mathrm{items}(cs)-\mathrm{items}(surr),\\ \mathsf{pos}_2:\mathrm{items}(surr)}} \arrow[d, dash] &&& \\
& \varrho_{\mathsf{pos}_2:\langle\mathsf{iter},\mathsf{ord},\mathsf{pos}\rangle} \arrow[d, dash] &&& \\
& \uplus \arrow[dl, dash] \arrow[drr, dash, dashed] &&& \\
@_{\mathsf{ord}:1} \arrow[d, dash] &&& \uplus \arrow[dl, dash] \arrow[dr, dash] & \\
q_1 && @_{\mathsf{ord}:n-1} \arrow[d, dash] && @_{\mathsf{ord}:n} \arrow[d, dash] \\
&& q_{n-1} && q_n
\end{tikzcd} \tag{RelAppend}$$

Table 3.8: The relational append operator concatenates the tuples of n queries.

3.6 Base Types

For any literal, such as a literal integer or string, the loop lifted encoding produces a table with a single element at position $\mathsf{pos} = 1$ for every iteration in *loop*. The translation rule for integers, decimals, strings and boolean values is depicted in rule (LL-LITERAL).

$$\frac{lit \in \{\texttt{int}[l], \texttt{dec}[l], \texttt{str}[l], \texttt{bool}[l]\}}{\Gamma; loop \vdash lit \Rrightarrow (@_{\mathsf{item}_1:l}(@_{\mathsf{pos}:1}(loop)), \mathsf{item}_1, \emptyset)} \text{ (LL-LITERAL)}$$

3.6.1 Binary and Unary Operators

SWITCH provides various infix operators on the values of base types ranging from simple arithmetics to the “Big Six” comparison operators. The translation rule (LL-BINARY) directly maps the infix operators (+, -, ...) to their relational equivalents (⊕, ⊖, ...).

The components e_1 and e_2 are translated individually to their loop-lifted translation, in a way to that in the rules for tuples and records. In this case, however, we expect a single numeric value in each iteration. The meta function $\curlyvee\ (q_1, q_2)$ (defined in Eq. (RelZip)) pairs tuples originating in the same iteration (the column item_1 in query q_2 is renamed to item_2 in order to prevent name clashes). As a result we expect a table presenting the operands side by side in columns item_1 and item_2 for each iteration. For each operator its relational equivalent is applied row-wise on the table to produce the overall result.

$$\frac{\begin{array}{c}\circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{|}, \texttt{\&}, \texttt{==}, \texttt{<>}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}\}\\ \Gamma; \mathit{loop} \vdash e_i \Rrightarrow (q_i, \mathsf{item}_1, \emptyset)\,\Big|_{i=1,2}\\ q \equiv \pi_{\mathsf{iter},\mathsf{pos},\mathsf{item}_3:\mathsf{item}_1}(\circledcirc_{\mathsf{item}_3:\langle \mathsf{item}_1,\mathsf{item}_2\rangle}(\curlyvee\ (q_1^1, q_2^1)))\end{array}}{\Gamma; \mathit{loop} \vdash \circ\texttt{(}e_1, e_2\texttt{)} \Rrightarrow (q, \mathsf{item}_1, \emptyset)} \quad \text{(LL-BINARY)}$$

In keeping with the above rule, the unary operators—the unary positive (`-`) and negative (`-`), as well as the logical negation (`!`)—are respectively mapped to their readily available primitives ($\oplus$, $\ominus$, ①) in the relational intermediate language.

$$\frac{\begin{array}{c}\circ \in \{\texttt{+}, \texttt{-}, \texttt{!}\}\\ \Gamma; \mathit{loop} \vdash e \Rrightarrow (q, \mathsf{item}_1, \emptyset)\\ q \equiv \pi_{\mathsf{iter},\mathsf{pos},\mathsf{item}_2:\mathsf{item}_1}(\circledcirc_{\mathsf{item}_2:\langle \mathsf{item}_1\rangle}(q))\end{array}}{\Gamma; \mathit{loop} \vdash \circ\texttt{(}e\texttt{)} \Rrightarrow (q, \mathsf{item}_1, \emptyset)} \quad \text{(LL-UNARY)}$$

3.7 Variables

Variables in SWITCH are merely introduced when using one of the iteration constructs in tandem with RUBY's block notation. The rules for these constructs provide the correct loop-lifted encoding for any variable x that is currently free in the iteration body, and populate the variable environment Γ accordingly with a plan bundle. When the variable x is accessed, its plan bundle is fetched from the variable environment and adopted as the resulting plan bundle.

$$\frac{\Gamma\, x = pb}{\Gamma; \mathit{loop} \vdash \texttt{var[}x\texttt{]} \Rrightarrow pb} \quad \text{(LL-VAR)}$$

3.8 Tuples and Records

Tuples, as well as records, wrap an arbitrary but finite number of elements into a combined type, while they remain accessible by referencing to either a position or, in case of records, to a label. The rule (LL-TUP) captures the loop lifting translation for a tuple. The intuition behind this rule is that we merge the derived loop-lifted encodings of all constituents into a single table by means of an equi-join.

$$\frac{\left.\begin{array}{ll} ① \ \Gamma; \mathit{loop} \vdash e_i \Rrightarrow (q_i, cs_i, surr_i) & ② \ l_i = |cs_i| \\ ③ \ cs'_i = cs_i \triangleright \sum_{j=1}^{i-1} l_j & ④ \ surr'_i = surr_i \triangleright \sum_{j=1}^{i-1} l_j \end{array}\right|_{i=1,\ldots,n}}{\Gamma; \mathit{loop} \vdash (\bar{e}) \Rrightarrow (\curlyvee (q_1^{l_1}, \ldots, q_n^{l_n}), (cs'_1, \ldots, cs'_n), \dot{\bigcup}_{i=1}^{n} surr'_i)} \text{(LL-TUP)}$$

In step ① of rule (LL-TUP) the elements e_1 to e_n are translated separately into their corresponding loop-lifted encodings respectively by employing the proper rule. Each of the queries (q_i) describes a table that adheres to the loop lifting schema $\mathbb{L}^{l_i}$ (comprising l_i item columns). The combined schema for the entire tuple is obtained by the total number of item columns of the individual queries and thus has the form $\mathbb{L}^{l_1+\cdots+l_n}$. The number of elements l_i for each query is calculated in step ②.

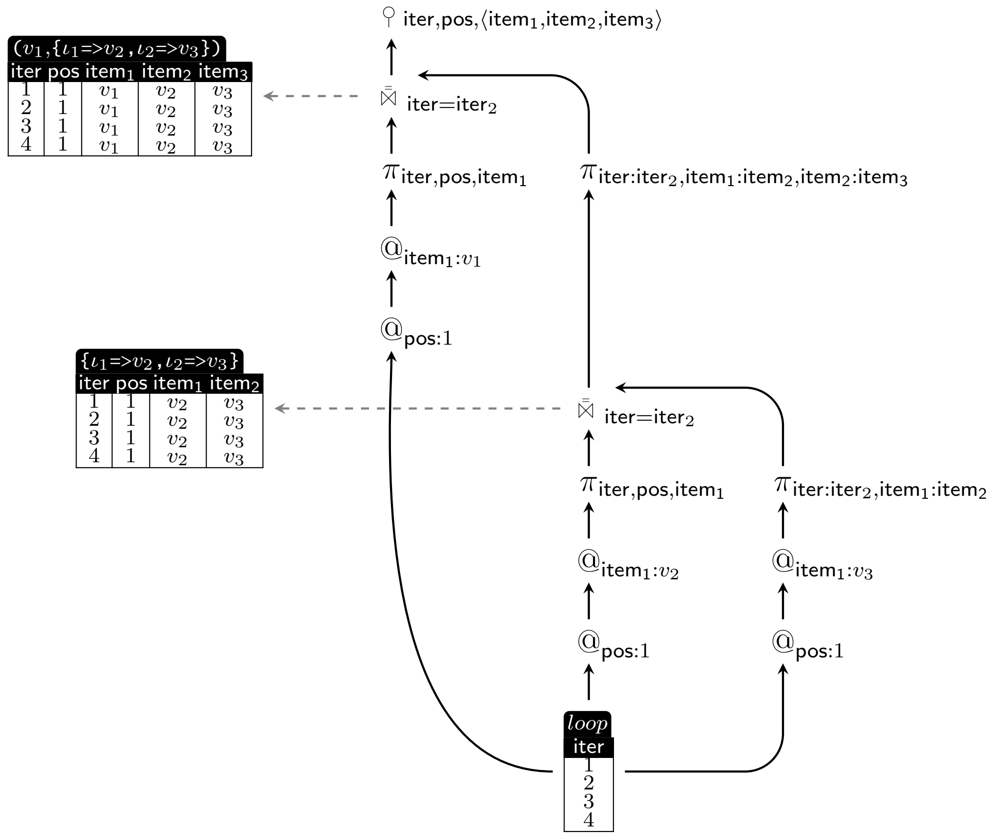
The problem that arises here is that the item sequence of each query (q_i) starts with item_1 and is consecutively numbered. Hence, the column names are mutually overlapping, which inevitably leads to name clashes. We prevent these name collisions by re-adjusting the item columns of each query so that they may safely be used side by side in a single table. In step ③ and step ④ we shift the item column names of each column structure cs_i and surrogate map $surr_i$ respectively by the overall number of item columns that are used to accommodate the preceding elements.

Having adjusted all auxiliary structures we must consider the queries (q_i), in which the same problem arises apparently. Here we rely on the meta function $\curlyvee (q_1^{l_1}, \ldots, q_n^{l_n})$ (see Eq. (RelZip)), which accepts the queries $q_1^{l_1}$ to $q_n^{l_n}$ jointly with the number of item columns and merges them into a single table with the schema $\mathbb{L}^{l_1+\cdots+l_n}$ by means of an equi-join on the iter column; item sequences are adjusted in the course of this process.

Rule (LL-TUPEMPTY) compiles the empty tuple into its tabular representation. Since this tuple has no components, both the column structure and surrogate map are empty.

$$\frac{}{\Gamma; \mathit{loop} \vdash () \Rrightarrow (@_{\mathsf{pos:1}}(\mathit{loop}), (), \emptyset)} \text{(LL-TUPEMPTY)}$$

Rule (LL-REC), which captures the translation of records, works analogously to the rule for tuples, with the exception that labels ι_1 to ι_n are employed to assemble the column structure to properly reflect the type. An illustration of how both rules elegantly interact to assemble a nested tuple out of the atomic values v_1 to v_3 is depicted in Figure 3.4.



ϱ iter,pos,$\langle$item$_1$,item$_2$,item$_3\rangle$
$\bowtie$ iter=iter$_2$
$\pi_{\text{iter,pos,item}_1}$
$\pi_{\text{iter:iter}_2\text{,item}_1\text{:item}_2\text{,item}_2\text{:item}_3}$
$@_{\text{item}_1:v_1}$
$@_{\text{pos:1}}$
$\bowtie$ iter=iter$_2$
$\pi_{\text{iter,pos,item}_1}$
$\pi_{\text{iter:iter}_2\text{,item}_1\text{:item}_2}$
$@_{\text{item}_1:v_2}$
$@_{\text{item}_1:v_3}$
$@_{\text{pos:1}}$
$@_{\text{pos:1}}$

(v₁,{ι₁=>v₂,ι₂=>v₃})

iter	pos	item$_1$	item$_2$	item$_3$
1	1	v_1	v_2	v_3
2	1	v_1	v_2	v_3
3	1	v_1	v_2	v_3
4	1	v_1	v_2	v_3

{ι₁=>v₂,ι₂=>v₃}

iter	pos	item$_1$	item$_2$
1	1	v_2	v_3
2	1	v_2	v_3
3	1	v_2	v_3
4	1	v_2	v_3

loop

iter
1
2
3
4

Figure 3.4: Loop lifted representation of the nested tuple $(v_1,\{\iota_1\text{=>}v_2,\iota_2\text{=>}v_3\})$ in the iteration context embodied by the loop table. Note how the item columns are properly renamed to prevent name clashes when two queries are paired.

$$\frac{\left.\begin{array}{ll}\Gamma; \mathit{loop} \vdash e_i \Rrightarrow (q_i, cs_i, surr_i) & l_i = |cs_i| \\ cs'_i = cs_i \rhd \sum_{j=1}^{i-1} l_j & surr'_i = surr_i \rhd \sum_{j=1}^{i-1} l_j\end{array}\right|_{i=1,\dots,n}}{\Gamma; \mathit{loop} \vdash \{\bar{\iota}\text{=>}\bar{e}\} \Rrightarrow (\curlyvee (q_1^{l_1}, \dots, q_n^{l_n}), \{\iota_1\colon cs'_1, \dots, \iota_n\colon cs'_n\}, \dot{\bigcup}_{i=1}^{n} surr'_i)} \text{ (LL-Rec)}$$

$$\frac{}{\Gamma; \mathit{loop} \vdash \{\} \Rrightarrow (@_{\text{pos:1}}(\mathit{loop}), \{\}, \emptyset)} \text{ (LL-RecEmpty)}$$

3.8.1 Positional and Nominal Reference

In rule (LL-TPos) we implement positional access on tuples ($[\cdot]^{\mathsf{T}}(e_{\text{tuple}},\texttt{int}[p])$) via position p. Note that the type system ensures that p does not exceed the number of elements in e_{tuple}. The rule is primarily concerned with compile-time bookkeeping tasks, whereas the query is expressed by means of a straightforward projection on the respective columns.

$$\dfrac{\begin{array}{l} \textcircled{1}\begin{cases} \Gamma; \mathit{loop} \vdash e_{\text{tuple}} \Rrightarrow (q,(cs_1,\dots,cs_p,\dots,cs_n),\mathit{surr}) \\ \{\mathsf{item}_{\mathsf{p_1}},\dots,\mathsf{item}_{\mathsf{p_m}}\} = \text{items}(cs_p)\end{cases} \\ \textcircled{2}\begin{cases} l = \sum_{i=1}^{p-1}|cs_i| \\ cs'_p = cs_p \lhd l \\ \mathit{surr}_p = \mathit{surr}\,|_{\text{items}(cs_p)} \lhd l\end{cases}\\ \textcircled{3}\ q' \equiv \pi_{\mathsf{iter,pos,item_{p_1}:item_1,\dots,item_{p_m}:item_m}}(q)\end{array}}{\Gamma;\mathit{loop}\vdash \texttt{[}\cdot\texttt{]}^{\mathsf{T}}\texttt{(}e_{\text{tuple}}\texttt{,int[}p\texttt{])} \Rrightarrow (q', cs'_p, \mathit{surr}_p)} \quad \text{(LL-TPos)}$$

In step ① the tuple (e_{tuple}) is translated into its tabular representation in the current iteration context. The remaining steps revolve around the column structure (cs_p), which maintains the item columns used to accommodate the referenced element within query q.

In step ② we restore the loop-lifted table schema: both column structure and surrogate map are properly aligned and confined based on the preceding item columns captured by the column structures (cs_1 through cs_{p-1}). In step ③ we eventually retrieve the element by a projection on its designated columns.

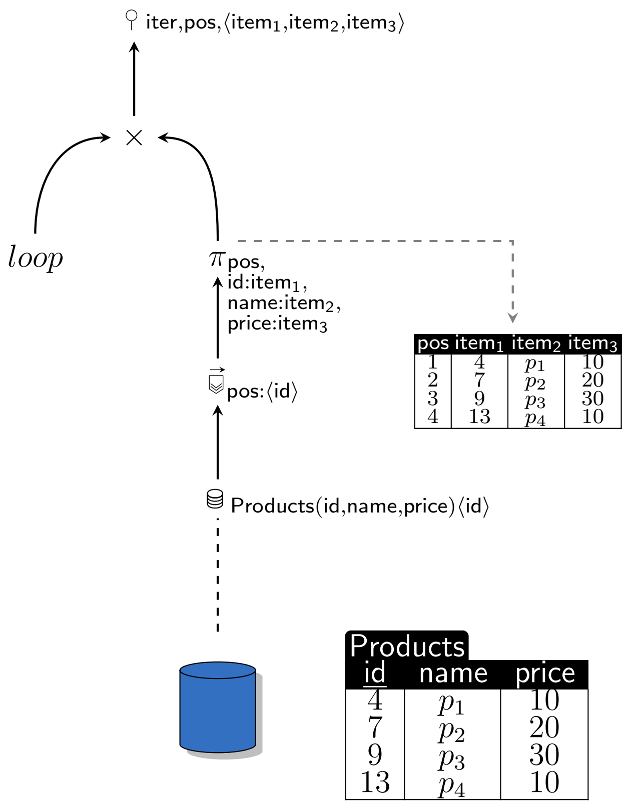
Nominal reference on a record captured by rule (LL-PROJ) operates analogous to rule (LL-TPOS). In this case, however, the projection is applied on label names rather than positions.

$$\dfrac{\begin{array}{l} \Gamma; \mathit{loop} \vdash e_{\text{record}} \Rrightarrow (q,\{\iota_1\colon cs_1,\dots,\iota_p\colon cs_p,\dots,\iota_n\colon cs_n\},\mathit{surr}) \\ \{\mathsf{item}_{\mathsf{p_1}},\dots,\mathsf{item}_{\mathsf{p_m}}\} = \text{items}(cs_p) \\ l = \sum_{i=1}^{p-1}|cs_i| \\ cs'_p = cs_p \lhd l \\ \mathit{surr}_p = \mathit{surr}\,|_{\text{items}(cs_p)} \lhd l \\ q' \equiv \pi_{\mathsf{iter,pos,item_{p_1}:item_1,\dots,item_{p_m}:item_m}}(q)\end{array}}{\Gamma;\mathit{loop}\vdash \texttt{proj(}e_{\text{record}}\texttt{,}\iota_p\texttt{)} \Rrightarrow (q', cs'_p, \mathit{surr}_p)} \quad \text{(LL-Proj)}$$

3.9 Interfacing the Relational Back-End

An important aspect of SWITCH is its ability to involve tables that reside in the underlying database back-end in an arbitrary computation. For this purpose, the language is furnished with a table reference operator (`table[T]`) that permits a table to be used as a source for constitutive manipulations. In SWITCH a table is represented as a list of records $[\{\iota_1\colon \tau_1,\dots,\iota_n\colon \tau_n\}]$:

(i) The labels ι_1 to ι_n are solely determined by the column names of the table,
(ii) while the respective column domain is straightforwardly mapped to the corresponding SWITCH types.



iter,pos,⟨item1,item2,item3⟩
×
loop
π pos, id:item1, name:item2, price:item3
pos:⟨id⟩
Products(id,name,price)⟨id⟩

pos	item$_1$	item$_2$	item$_3$
1	4	p_1	10
2	7	p_2	20
3	9	p_3	30
4	13	p_4	10

Products

id	name	price
4	p_1	10
7	p_2	20
9	p_3	30
13	p_4	10

Figure 3.5: The query plan accesses the Products table hosted by the underlying database back-end (id denotes the primary key). The table shown on the right taps the intermediate result before the cross-product with the iteration context is applied.

(iii) To establish a logical order between elements in the list, we set up a position column based on the primary key (the actual choice of a sorting criterion, however, is arbitrary).

In rule (LL-Table) we comprise the above steps in a single rule. The column names are preserved in the column structure whereas the table schema is entirely renamed to impose the loop-lifted table schema to the query component. As a last step, we establish the iteration context by use of a cross product (for an illustration consider Figure 3.5).

$$\frac{q \equiv loop \times \pi_{\mathsf{pos},c_1:\mathsf{item}_1,\ldots,c_n:\mathsf{item}_n}\left(\vec{\varrho}_{pos:\langle p\rangle}\left(\mathsf{T}_{(c_1,\ldots,c_n)\langle p\rangle}\right)\right)}{\Gamma; loop \vdash \texttt{table[T]} \Rrightarrow (q, \{c_1\colon \mathsf{item}_1,\ldots,c_n\colon \mathsf{item}_n\}, \emptyset)} \quad \text{(LL-Table)}$$

3.10 Nesting

The handling of nested lists is of critical importance in SWITCH. So far, we have only seen how tuples and records may be nested accordingly to be rendered into a single row. Naturally, a list spans several rows that are tied by its list identifier. Therefore we must distribute nested lists over several flat tables linked by surrogate values to properly present them to the database back-end.

For illustration, examine the following nested list that may be introduced in the expression tree of a SWITCH program:

`<<10, 20, 30>, <40, 50>>` (Q_3)

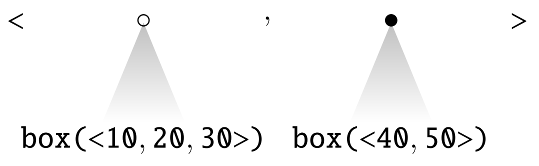
The list constructor features two further lists that jointly span exactly five rows in their tabular encoding. Consequently, the expression in its entirety cannot reside in a single row and has to enriched with information that explicitly tells us to

(i) split its tabular representation into two queries ($Q_{3\cdot 1}$ and $Q_{3\cdot 2}$)
(ii) and to introduce surrogate values which relate the queries by a foreign-key relationship in order to restore the original list.

For this purpose we introduce

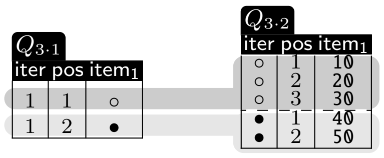
`box(`e`)` ,

which, weaved into the expression tree, forks the compilation to translate two relational queries to properly present Query Q_3 to its flat table operators. The expression below demonstrates how **box(**e**)** helps in the compilation of the above expression:



Now, with the information that the inner lists are respectively represented by a single value that occupies a table cell, the list constructor is merely concerned with computing the tabular encoding of the outer list (where all inner lists have been substituted by their corresponding surrogates). Note that the surrogate value of an empty list does not appear in the inner query.

The inner lists are then assembled into a single table, where the surrogate values are used to discriminate between them. Here, the `iter` column, hosting the surrogate values, additionally assumes the role of a list identifier, resulting in the tables in Figure 3.6.



$Q_{3\cdot 1}$

iter	pos	item$_1$
1	1	○
1	2	●

$Q_{3\cdot 2}$

iter	pos	item$_1$
○	1	10
○	2	20
○	3	30
●	1	40
●	2	50

Figure 3.6: Tabular representation of Query Q_3. Note how the inner lists are combined in a single table and may be easily discriminated according to their iter column.

A similar problem arises when we use the following query to obtain the total over the inner lists (for the sake of simplicity we substituted the binding list by a literal as it would be seen by SWITCH):

```
map(<<10,20,30>,<40,50>>,λx.sum(var[x]))   (Q4)
  # ⇝ <60,90>
```

Whereas `box()` is introduced to respectively condense the inner lists to a surrogate, the computation in the body expects the loop variable to present these lists—and its values—in a single table to successfully complete its task. In a sense, we need to roll back the box operation on the iteration variable to properly calculate the total.

To swap between the tabular and the row-wise list representation, we weave

```
unbox(e)
```

into the expression tree, resulting in the following query that has been properly enriched with the respective operations in order to ensure a flat representation of all participating constructs:

```
map(<box(<10,20,30>),box(<40,50>)>,λx.sum(unbox(var[x])))
```

The `unbox()` operation eventually triggers the compilation process to combine the two queries ($Q_{3\cdot 1}$and $Q_{3\cdot 2}$) that have been introduced by the `box()` operation. Later on, we will see how this is efficiently obtained by an equi-join on the surrogate columns.

3.10.1 Switch between Representations

In terms of a database back-end, any SWITCH expression may be either represented by a single row or, in case of lists, by multiple rows tied by their list identifier. Based on this rough differentiation, the types in Table 3.9 classify any expression according to their representation in the underlying back-end. In the following we refer to them as *implementation types*:

Implementation Types

β, γ, δ	::=	*Atom*	single row
	|	*List*	multiple rows

Table 3.9: The above types classify any expression based on how they are implemented in the underlying database back-end.

Atom Every expression that may be rendered into a single row ends up in this category. We define *Atom* inductively over the type of a SWITCH expression:

$$\begin{array}{rrl} & \mathit{Int}, \mathit{Str}, \mathit{Dec}, \mathit{Bool} & \in \mathit{Atom} \\ \text{if } \tau_i \in \mathit{Atom} \text{ for all } i = 1, \ldots, n \text{ then} & (\tau_1, \ldots, \tau_n) & \in \mathit{Atom} \\ \text{if } \tau_i \in \mathit{Atom} \text{ for all } i = 1, \ldots, n \text{ then} & \{\iota_1 \colon \tau_1, \ldots, \iota_n \colon \tau_n\} & \in \mathit{Atom} \end{array}$$

The above definition tells us that base types are elements of *Atom* by definition. Furthermore, any arbitrary nested tuple or record may be implemented by a single row as long as the types of its direct constituents are elements of *Atom* as well.

List The second category tells us whether an expression is represented by a group of rows in a table, and therefore, merely populated by lists (of arbitrary type):

$$\forall \tau, [\tau] \in \mathit{List}$$

Following the above definitions, these coarse types may be easily derived based on the type system we provide in Section 2.6. In Table 3.10 we applied the definitions to summarize the built-in functions in terms of their implementation types.

The type system summarized by the set inference rules in Table 3.11 performs a simple bottom-up static analysis that precedes the algebraic compilation. Along with the proper implementation type for an expression, it prescribes the introduction of runtime coercions (`box(`e`)` and `unbox(`e`)`), *i.e.* an expression e does not meet the expected type. The judgment $e \rightarrow e' : \beta$ reads as "The expression e is possibly coerced into e' to meet the expected implementation type, and has implementation type β."

In the inference rules, we use the following operator that triggers the introduction of `box(`e`)` or `unbox(`e`)`:

$$⇅^{\beta}_{\gamma}(e) = \begin{cases} e & \text{, if } \beta = \gamma \\ \texttt{box}(e) & \text{, if } \beta = \mathit{List} \text{ and } \gamma = \mathit{Atom} \\ \texttt{unbox}(e) & \text{, if } \beta = \mathit{Atom} \text{ and } \gamma = \mathit{List} \end{cases}$$

By applying the inference rules on an expression e we obtain an expression, that is enriched with runtime coercions, whose loop-lifted translations we devise in the following section.

Built-in functions

$\texttt{concat}^\mathsf{L} : \mathit{List} \times \mathit{List} \rightarrow \mathit{List}$	concatenation
$\texttt{[}\cdot\texttt{]}^\mathsf{T} : \mathit{Atom} \times \mathit{Atom} \rightarrow \mathit{Atom}$	positional access
$\texttt{[}\cdot\texttt{]}^\mathsf{L} : \mathit{List} \times \mathit{Atom} \rightarrow \mathit{Atom}$	positional access
$\texttt{first}^\mathsf{L} : \mathit{List} \rightarrow \mathit{Atom}$	first element
$\texttt{last}^\mathsf{L} : \mathit{List} \rightarrow \mathit{Atom}$	last element
$\texttt{take}^\mathsf{L} : \mathit{List} \times \mathit{Atom} \rightarrow \mathit{List}$	keep prefix
$\texttt{drop}^\mathsf{L} : \mathit{List} \times \mathit{Atom} \rightarrow \mathit{List}$	keep suffix
$\texttt{reverse}^\mathsf{L} : \mathit{List} \rightarrow \mathit{List}$	reversal
$\texttt{length}^\mathsf{L} : \mathit{List} \rightarrow \mathit{Atom}$	list length
$\texttt{flatten} : \mathit{List} \rightarrow \mathit{List}$	list flattening
$\texttt{sum}, \texttt{avg}, \texttt{min}, \texttt{max} : \mathit{List} \rightarrow \mathit{List}$	list aggregation
$\texttt{member?} : \mathit{List} \times \mathit{Atom} \rightarrow \mathit{Atom}$	element lookup
$\texttt{uniq} : \mathit{List} \rightarrow \mathit{List}$	duplicate elimination
$\texttt{zip} : \mathit{Atom} \rightarrow \mathit{List}$	zip
$\texttt{unzip} : \mathit{List} \rightarrow \mathit{Atom}$	unzip

Higher order built-in functions

$\texttt{map} : \mathit{List} \times (\mathit{Atom} \rightarrow \mathit{Atom}) \rightarrow \mathit{List}$	iterate over elements
$\texttt{select}, \texttt{reject} : \mathit{List} \times (\mathit{Atom} \rightarrow \mathit{Atom}) \rightarrow \mathit{List}$	filter elements
$\texttt{flat\_map} : \mathit{List} \times (\mathit{Atom} \rightarrow \mathit{Atom}) \rightarrow \mathit{List}$	iteration and flattening
$\texttt{all?}, \texttt{any?} : \mathit{List} \times (\mathit{Atom} \rightarrow \mathit{Atom}) \rightarrow \mathit{Atom}$	quantification
$\texttt{take\_while}, \texttt{drop\_while} : \mathit{List} \times (\mathit{Atom} \rightarrow \mathit{Atom}) \rightarrow \mathit{List}$	prefix and suffix
$\texttt{count} : \mathit{List} \times (\mathit{Atom} \rightarrow \mathit{Atom}) \rightarrow \mathit{List}$	count elements
$\texttt{sort\_by} : \mathit{List} \times (\mathit{Atom} \rightarrow \mathit{Atom}) \rightarrow \mathit{List}$	sorting
$\texttt{min\_by}, \texttt{max\_by} : \mathit{List} \times (\mathit{Atom} \rightarrow \mathit{Atom}) \rightarrow \mathit{Atom}$	minimum and maximum
$\texttt{partition} : \mathit{List} \times (\mathit{Atom} \rightarrow \mathit{Atom}) \rightarrow \mathit{List}$	partition
$\texttt{group\_with} : \mathit{List} \times (\mathit{Atom} \rightarrow \mathit{Atom}) \rightarrow \mathit{List}$	grouping

Table 3.10: Implementation types of built-in functions

Table 3.11: Static analysis that prescribes the introduction of runtime coercions `box(`e`)` and `unbox(`e`)`, if *List*/*Atom* mismatches are encountered in e (or its sub-expressions).

$$\frac{}{\texttt{int[}i\texttt{]} \rightarrow \texttt{int[}i\texttt{]} : \mathit{Atom}}\ (\textsc{Box-Int}) \qquad \frac{}{\texttt{dec[}d\texttt{]} \rightarrow \texttt{dec[}d\texttt{]} : \mathit{Atom}}\ (\textsc{Box-Dec})$$

$$\frac{}{\texttt{str[}\mathrm{s}\texttt{]} \rightarrow \texttt{str[}\mathrm{s}\texttt{]} : \mathit{Atom}}\ (\textsc{Box-Str}) \qquad \frac{}{\texttt{bool[}b\texttt{]} \rightarrow \texttt{bool[}b\texttt{]} : \mathit{Atom}}\ (\textsc{Box-Bool})$$

$$\frac{}{\texttt{var[}x\texttt{]} \rightarrow \texttt{var[}x\texttt{]} : \mathit{Atom}}\ (\textsc{Box-Var})$$

$$\frac{}{\texttt{()} \rightarrow \texttt{()} : \mathit{Atom}}\ (\textsc{Box-TupEmpty})$$

$$\frac{e_i \rightarrow e'_i : \beta_i \Big|_{i=1,\ldots,n}}{\texttt{(}\bar{e}\texttt{)} \rightarrow \texttt{(}\boxed{\downarrow}^{\beta_1}_{\mathit{Atom}}(e'_1), \ldots, \boxed{\downarrow}^{\beta_n}_{\mathit{Atom}}(e'_n)\texttt{)} : \mathit{Atom}}\ (\textsc{Box-TupCons})$$

$$\frac{}{\texttt{\{\}} \rightarrow \texttt{\{\}} : \mathit{Atom}}\ (\textsc{Box-RecEmpty})$$

$$\frac{e_i \rightarrow e'_i : \beta_i \Big|_{i=1,\ldots,n}}{\texttt{\{}\bar{\iota}\texttt{=>}\bar{e}\texttt{\}} \rightarrow \texttt{\{}\iota_1\texttt{=>}\boxed{\downarrow}^{\beta_1}_{\mathit{Atom}}(e'_1), \ldots, \iota_n\texttt{=>}\boxed{\downarrow}^{\beta_n}_{\mathit{Atom}}(e'_n)\texttt{\}} : \mathit{Atom}}\ (\textsc{Box-RecCons})$$

$$\frac{}{\texttt{<>} \rightarrow \texttt{<>} : \mathit{List}}\ (\textsc{Box-EmptyList})$$

$$\frac{e_i \rightarrow e'_i : \beta_i \Big|_{i=1,\ldots,n}}{\texttt{<}\bar{e}\texttt{>} \rightarrow \texttt{<}\boxed{\downarrow}^{\beta_1}_{\mathit{Atom}}(e'_1), \ldots, \boxed{\downarrow}^{\beta_n}_{\mathit{Atom}}(e'_n)\texttt{>} : \mathit{List}}\ (\textsc{Box-ListCons})$$

$$\frac{e \rightarrow e' : \mathit{Atom}}{\texttt{proj(}e\texttt{,}\iota\texttt{)} \rightarrow \texttt{proj(}e'\texttt{,}\iota\texttt{)} : \mathit{Atom}}\ (\textsc{Box-RecProj})$$

continued on next page

$$\frac{}{\texttt{table[R]} \rightarrow \texttt{table[R]} : \mathit{List}} \quad \text{(Box-TableRef)}$$

$$\frac{e_1 \rightarrow e_1' : \mathit{Atom} \qquad e_i \rightarrow e_i' : \beta\,\Big|_{i=2,3}}{\texttt{if(}e_1\texttt{,}e_2\texttt{,}e_3\texttt{)} \rightarrow \texttt{if(}e_1'\texttt{,}e_2'\texttt{,}e_3'\texttt{)} : \beta} \quad \text{(Box-If)}$$

$$\frac{e_i \rightarrow e_i' : \mathit{Atom}\,\Big|_{i=1,2}}{\circledast\texttt{(}e_1, e_2\texttt{)} \rightarrow \circledast\texttt{(}e_1', e_2'\texttt{)} : \mathit{Atom}} \quad \text{(Box-Arith)}$$

$$\frac{e_i \rightarrow e_i' : \mathit{Atom}\,\Big|_{i=1,2}}{\olessthan\texttt{(}e_1, e_2\texttt{)} \rightarrow \circledast\texttt{(}e_1', e_2'\texttt{)} : \mathit{Atom}} \quad \text{(Box-Comp)}$$

$$\frac{e_i \rightarrow e_i' : \mathit{Atom}\,\Big|_{i=1,2}}{\owedge\texttt{(}e_1, e_2\texttt{)} \rightarrow \circledast\texttt{(}e_1', e_2'\texttt{)} : \mathit{Atom}} \quad \text{(Box-Junc)}$$

$$\frac{\begin{array}{c}\Gamma\, m : \beta_1 \times \ldots \times \beta_n \rightarrow \gamma. \\ e_i \rightarrow e_i' : \delta_i\,\Big|_{i=1,\ldots,n}\end{array}}{m\texttt{(}\bar{e}\texttt{)} \rightarrow m\texttt{(}\boxed{\downarrow}^{\delta_1}_{\beta_1}(e_1'), \ldots, \boxed{\downarrow}^{\delta_n}_{\beta_n}(e_n')\texttt{)} : \gamma} \quad \text{(Box-Func)}$$

$$\frac{\begin{array}{c}\Gamma\, m : \mathit{List}_\bullet \times (\mathit{Atom} \rightarrow \mathit{Atom}_\bullet) \rightarrow \beta \\ e \rightarrow e' : \gamma \qquad\qquad e_b \rightarrow e_b' : \delta\end{array}}{m\texttt{(}e, \lambda x.e_b\texttt{)} \rightarrow m\texttt{(}\boxed{\downarrow}^{\gamma}_{\mathit{List}_\bullet}(e'), \lambda x.\boxed{\downarrow}^{\delta}_{\mathit{Atom}_\bullet}(e_b')\texttt{)} : \beta} \quad \text{(Box-FuncBlock)}$$

3.10.2 (Un)Box

To split a query representing an expression e into two relating queries we have recourse on the iteration context in which e is compiled. Since *loop* holds a unique value for each iteration it may be safely used to establish foreign key relationship

$$\pi_{\mathsf{item}_1}(q') \supseteq \pi_{\mathsf{iter}}(q) \ ,$$

whereby the original expression may be restored.

Note how query q' is used in rule (LL-Box) to substitute the tabular encoding of expression e. The current iteration context participates in the preparation of a surrogate column (item_1) to be inserted into the surrogate map, in tandem with the entire plan bundle derived from the compilation of e.

$$\frac{\Gamma; \mathit{loop} \vdash e \Rrightarrow (q_{\text{inner}}, cs, \mathit{surr}) \qquad q_{\text{outer}} \equiv @_{\mathsf{pos}:1}(\pi_{\mathsf{iter}:\{\mathsf{iter},\mathsf{item}_1\}}(\mathit{loop}))}{\Gamma; \mathit{loop} \vdash \mathtt{box}(e) \Rrightarrow (q_{\text{outer}}, \mathsf{item}_1, \{\mathsf{item}_1 \mapsto (q_{\text{inner}}, cs, \mathit{surr})\})} \text{(LL-Box)}$$

In rule (LL-UNBOX) we provide the inference rule to compile $\mathtt{unbox}(e)$ that merges two queries that jointly implement a nested expression e into a single query. The equi-join operator, applied to the surrogate columns, efficiently implements this operation.

$$\frac{\Gamma; \mathit{loop} \vdash e \Rrightarrow (q_{\text{outer}}, \mathsf{item}_1, \{\mathsf{item}_1 \mapsto (q_{\text{inner}}, cs, \mathit{surr})\}) \qquad q \equiv \pi_{\mathsf{iter}_2:\mathsf{iter},\mathsf{pos},\text{items}(cs)}(\pi_{\mathsf{iter}:\mathsf{iter}_2,\mathsf{item}_1:\mathsf{surr}}(q_{\text{outer}}) \overset{=}{\bowtie}_{\mathsf{surr}=\mathsf{iter}} q_{\text{inner}})}{\Gamma; \mathit{loop} \vdash \mathtt{unbox}(e) \Rrightarrow (q, cs, \mathit{surr})} \text{(LL-Unbox)}$$

3.10.3 Avoiding Query Avalanches

SWITCH guarantees that only few queries are fired against the database back-end. This marks a significant deviation from ACTIVERECORD, which still suffers from the n+1 query problem and keeps the back-end busy with a flood of simple look-alike queries. Whereas ACTIVERECORD emits queries dependent on the database instance size, in SWITCH it is exclusively the expression's static result type that determines the number of initiated SQL queries. More specific, the number of queries #queries(τ) corresponds to the sum of the nesting depth of an expression with the type τ (see also [GG+13]):

$$\#\text{queries}(\tau) = \begin{cases} 1 & \text{, if } \tau \in \mathit{Atom} \\ \text{sumdep}(\tau) & \text{, else} \end{cases}$$

with sumdep(τ) defined as follows

$$\text{sumdep}(\tau) = \begin{cases} 0 & \text{, if } \tau \in \{\mathit{Int}, \mathit{Dec}, \mathit{Str}, \mathit{Bool}\} \\ \text{sumdep}(\tau_1) + \ldots + \text{sumdep}(\tau_n) & \text{, if } \tau = (\tau_1, \ldots, \tau_n) \\ \text{sumdep}(\tau_1) + \ldots + \text{sumdep}(\tau_n) & \text{, if } \tau = \{\iota_1 : \tau_1, \ldots, \iota_n : \tau_n\} \\ 1 + \text{sumdep}(\sigma) & \text{, if } \tau = [\sigma] \end{cases}$$

The above definition for #queries() tells us that for an atom (Atom) SWITCH issues a single query against the database in order to derive the result. In any other case—if list types occur in τ—the nesting depth is determined via sumdep(τ). For example, an expression with the result type $[([[\mathit{Int}]], [\mathit{Str}])]$ eventually leads to four query plans that are respectively translated into four SQL queries by the SQL generator.

The reasons for this behavior lie in the type system in Table 3.11, which introduces the operations `box()` and `unbox()` in order to cope with nesting. As we detailed in the preceding sections, these operations are introduced based on the types of the participating constructs of an expression. The operation `box()` (rule (LL-Box)) increases the number of initiated queries by splitting the loop-lifted encoding of an expression, whereas the operation `unbox()` (rule (LL-Unbox)) merges two queries into a single one and therefore decreases the number of queries.

3.11 Surrogate Maps: A Home for Nested Lists

Surrogate maps constitute an important part of our compilation scheme because they accommodate the loop-lifted encodings of nested expressions. The *compile-time operations* we describe in the current section originate in the observation that operations applied to lists—like `concat()` or `drop()`—can also affect nested items. The following meta operations assure that list manipulations, performed on an outer list, are also propagated to the nested list expressions.

3.11.1 Appending Nested Lists

In the preceding section we have seen how Query Q_3 was enriched by `box()` operations in order to represent the inner lists by means of surrogate values.

$$\texttt{<box(<10, 20, 30>), box(<40, 50>)>}$$

Due to the compositionality of our compilation rules, each inner list is compiled separately and results in a plan bundle of its own:

$$\Gamma; loop \vdash \texttt{box(<10, 20, 30>)} \Rrightarrow (q_{1\cdot 1}, \mathsf{item}_1, \overbrace{\{\mathsf{item}_1 \mapsto (q_{1\cdot 2}, \mathsf{item}_1, \emptyset)\}}^{\mathsf{surr}_1})$$
$$\Gamma; loop \vdash \texttt{box(<40, 50>)} \Rrightarrow (q_{2\cdot 1}, \mathsf{item}_1, \underbrace{\{\mathsf{item}_1 \mapsto (q_{2\cdot 2}, \mathsf{item}_1, \emptyset)\}}_{\mathsf{surr}_2})$$

The compilation rule for the list constructor (rule (LL-List) on page 87) is primarily concerned with merging the "outer queries" ($q_{1\cdot 1}$ and $q_{2\cdot 1}$) into a single query by means of a disjoint union (via Eq. (RelAppend)). Note that $q_{1\cdot 1}$ and $q_{2\cdot 1}$ only contain surrogate values whereas the surrogate maps (surr_1 and surr_2) contain the queries ($q_{1\cdot 1}$ and $q_{2\cdot 1}$) that derive the actual data for the inner lists. Hence, to properly conclude the translation of the list constructor we must combine the queries in the surrogate maps.

The meta operation

$$\llbracket surr_1, \ldots, surr_n \rrbracket_q^{\uplus}$$

in its general form, combines the surrogate maps ($surr_1$ through $surr_n$) and results in a single surrogate map by respectively combining the query plans that originate in the same surrogate column. Note how Eq. (Meta-Append) iterates through the surrogate map to concatenate the queries by means of a disjoint union (see Eq. (RelAppend)). The operation is then recursively applied to all "nested" surrogate maps ($surr_{1\cdot i}$ through $surr_{n\cdot i}$) in order to consider further nesting levels. A detailed example that involves this operation can be found in Figure 3.8, which shows how inner lists are merged into a single query plan.

$$\llbracket surr_1, \ldots, surr_n \rrbracket^{\uplus}_{q} =$$

$$\left\{ \begin{array}{rl|l} \textbf{let}\ (q_{1\cdot i}, cs, surr_{1\cdot i}) = & surr_1\ \mathsf{item_i}\, , & \\ & \vdots & \\ (q_{n\cdot i}, cs, surr_{n\cdot i}) = & surr_n\ \mathsf{item_i}\, , & \\ q_{\text{app}} = & \boxplus^{(cs, surr_{1\cdot i})}(q_{1\cdot i}, \ldots, q_{n\cdot i})\, , & \mathsf{item_i} \in dom(surr_1) \\ q' = & \pi_{\mathsf{iter:iter_2,ord:ord_2,pos:pos_2}}(q)\, , & \\ q'_{\text{app}} = & q_{\text{app}} \overset{=}{\bowtie}_{\mathsf{iter=iter_2 \land ord=ord_2}}\, q'\, , & \\ q_i = & \pi_{\mathsf{pos_2:iter,pos,}\mathrm{items}(cs)}(q'_{\text{app}}) & \\ \multicolumn{2}{l|}{\textbf{in}\ \mathsf{item_i} \mapsto (q_i, cs, \llbracket surr_{1\cdot i}, \ldots, surr_{n\cdot i} \rrbracket^{\uplus}_{q_{\text{app}}})} & \end{array} \right\}$$

(Meta-Append)

Table 3.12: The above rules recursively append the query plans in the surrogate map ($surr_1$ through $surr_n$) by means of a disjoint union. The query q has the schema $\langle \mathsf{iter}, \mathsf{pos}, \mathsf{ord}, \overline{\mathsf{item}} \rangle$. Note that the column pos (temporarily renamed to $\mathsf{pos_2}$) provides the new iter column for the "nested queries" (q_i) in order to link them to the "outer queries".

3.11.2 Removing Nested Lists

If in the course of a computation elements are dropped from a list, this removal needs to be established in their nested elements. For an example, consider the following query that drops the first element from a nested list:

```
dropᴸ(<<10,20,30>,<40,50>>,1)     (Q5)
  # ⇝ <<40,50>>
```

To propagate the removal of the first element to its nested data structures we rely on the rule summarized by Table 3.13. This compile-time operation arranges a new query plan that clears the surrogate maps of elements that have not been removed outright with the originating element. Query q in the rules depicted in Table 3.13 contains the iteration identifiers in a single row together with the surrogates that can be found in the queries of the surrogate map. We use an equi-join to discard all elements not stemming from the iteration context described by query q: surrogates that are dropped do not find a matching value. This schema is recursively reproduced to clear the entire plan bundle of leftover rows. For a demonstrating example of how the loop-lifted encoding of Query Q_5 is affected consider Figure 3.7.

$$\llbracket surr \rrbracket^{\curlywedge}_{q} = \left\{ \begin{array}{l} \textbf{let } (q_i, cs_i, surr_i) = surr\ \mathsf{item_i}\,\textbf{,} \\ \qquad\qquad q'_i = \text{restrict}(\pi_{\mathsf{item_i:iter}}(q), q_i^{|cs_i|}) \\ \ \textbf{in } \mathsf{item_i} \mapsto (q'_i, cs_i, \llbracket surr_i \rrbracket^{\curlywedge}_{q'_i}) \end{array} \middle| \ \mathsf{item_i} \in dom(surr) \right\} \tag{Meta-Filter}$$

Table 3.13: Propagates the removal of elements in query q to the nested data in the surrogate maps. The query q contains surrogate values in the columns $\mathsf{item_i}$ that we use in order to refer to the plan bundles of the nested expressions in the surrogate map (surr).

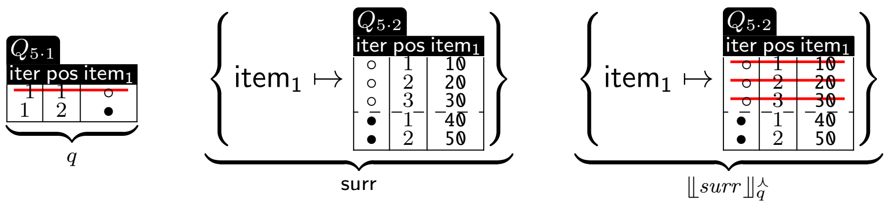


Figure 3.7: The above figure demonstrates how the surrogate map is affected, when Query Q_5 removes the first item from the nested input list. Recall that the surrogate map (surr) hosts the tabular encoding of the list items, each of which being a nested inner lists. In this example, $\llbracket \cdot \rrbracket^{\curlywedge}_{\mathrm{q}}$ faithfully discards the first inner list from the surrogate map. The red rules indicate dropped rows.

3.12 Conditionals

The compilation of conditionals is quite complex and involves several steps summarized in the single inference rule (LL-IF):

In step ① we translate the expression e_{bool} in the current iteration context (e_{bool} evaluates to a boolean value). The resulting query q_{bool} is then used in step ② to divide the iteration context into two partitions, described by $loop_{\text{then}}$ and $loop_{\text{else}}$. Based on these new iteration contexts, we establish separate variable environments (Γ_{then} and Γ_{else}) by confining the loop-lifted representations of the free variables in the environment Γ to iterations of the respective *then*-branch or *else*-branch (see Eq. (Env-Restrict)). The subexpressions e_{then} and e_{else} are then translated using the respective iteration context and variable environment.

In step ③ the resulting queries (q_{then} and q_{else}) are merged into a single query (q_{if}) by means of a disjoint union in order to form the overall result (consider Eq. (RelAppend)). Note that we use $\llbracket\cdot\rrbracket^{\boxplus}_{\mathsf{q}}$ to reproduce this step throughout (possibly) nested lists hosted by the surrogate maps.

$$\dfrac{\begin{array}{l}
① \; \Gamma; loop \vdash e_{\text{bool}} \Rrightarrow (q_{\text{bool}}, \mathsf{item_1}, \emptyset) \\[1em]
② \begin{cases}
\begin{array}{l|l}
\qquad\qquad \textit{then}\text{-branch} & \qquad\qquad \textit{else}\text{-branch} \\
loop_{\text{then}} \equiv \pi_{\mathsf{iter}}(\sigma_{\mathsf{item_1}}(q_{\text{bool}})) & loop_{\text{else}} \equiv \pi_{\mathsf{iter}}(\sigma_{\neg\mathsf{item}}(q_{\text{bool}})) \\
\Gamma_{\text{then}} \equiv (\!|\Gamma|\!)^{\curlywedge}_{loop_{\text{then}}} & \Gamma_{\text{else}} \equiv (\!|\Gamma|\!)^{\curlywedge}_{loop_{\text{else}}} \\
\Gamma_{\text{then}}; loop_{\text{then}} \vdash e_{\text{then}} \Rightarrow & \Gamma_{\text{else}}; loop_{\text{else}} \vdash e_{\text{else}} \Rightarrow \\
\qquad\qquad (q_{\text{then}}, cs_{\text{if}}, surr_{\text{then}}) & \qquad\qquad (q_{\text{else}}, cs_{\text{if}}, surr_{\text{else}})
\end{array}
\end{cases} \\[1em]
③ \begin{cases}
q \equiv \boxplus^{(cs_{\text{if}}, surr_{\text{then}})}(q_{\text{then}}, q_{\text{else}}) \\
q_{\text{if}} \equiv \pi_{\mathsf{iter},\mathsf{pos},\text{items}(cs_{\text{if}})}(q) \\
surr_{\text{if}} = \llbracket surr_{\text{then}}, surr_{\text{else}} \rrbracket^{\boxplus}_{q}
\end{cases}
\end{array}}{\Gamma; loop \vdash \texttt{if(}e_{\text{bool}}\texttt{,}e_{\text{then}}\texttt{,}e_{\text{else}}\texttt{)} \Rrightarrow (q_{\text{if}}, cs_{\text{if}}, surr_{\text{if}})} \quad \text{(LL-IF)}$$

3.13 Lists

Recall that the list constructor is merely used internally when a tuple is promoted to a list to meet typing rules; in the surface syntax it is not possible to directly assemble a list. Like the tuple constructor, the list constructor accepts an arbitrary but finite number of elements. A list, however, requires all its elements to be equally typed. Rule (LL-LIST) compiles a list into its loop-lifted encoding.

Initially, in step ① all direct constituents are translated into their corresponding tabular representation. In step ② we have recourse to these derived queries (q_1

to q_n) in order to concatenate them via Eq. (RelAppend). The resulting query q participates in the preparation of the surrogates maps for the final result.

In step ③ the surrogate maps are traversed via the meta function $\llbracket\cdot\rrbracket^{\uplus}_{q}$ to account possibly nested lists. This operation recursively appends the query plans in the surrogate maps ($surr_1$ through $surr_n$) and results in the single surrogate map $surr$.

$$\frac{\begin{array}{l} ①\ \Gamma; loop \vdash e_i \Rrightarrow (q_i, cs, surr_i)\big|_{i=1,\ldots,n} \\ ②\ q \equiv \uplus^{(cs,surr_1)}(q_1,\ldots,q_n) \\ ③\ surr = \llbracket surr_1,\ldots,surr_n\rrbracket^{\uplus}_{q} \end{array}}{\Gamma; loop \vdash \texttt{<}\bar{e}\texttt{>} \Rrightarrow (\pi_{\mathsf{iter,pos,items}(cs)}(q), cs, surr)} \quad \text{(LL-LIST)}$$

The rule (LL-LCONCAT) captures the concatenation of two lists. Concatenation expects its arguments (e_1 and e_2) to be equally typed lists whereas the list constructor handles elements of any type as long as their types are equal. The below rule is a reprisal of the more general rule above and operates analogously except for its confinement on the number of arguments.

$$\frac{\begin{array}{l} \Gamma; loop \vdash e_1 \Rrightarrow (q_i, cs, surr_i)\big|_{i=1,2} \\ q \equiv \uplus^{(cs,surr_1)}(q_1,q_2) \\ surr = \llbracket surr_1, surr_2\rrbracket^{\uplus}_{q} \end{array}}{\Gamma; loop \vdash \texttt{concat}^{\mathsf{L}}\texttt{(}e_1\texttt{,}e_2\texttt{)} \Rrightarrow (\pi_{\mathsf{iter,pos,items}(cs)}(q), cs, surr)} \quad \text{(LL-LCONCAT)}$$

3.13.1 Positional Access

To access an element of a list based on its position, consider the following query, which obtains the second element in the Products table (the primary key prescribes the arrangement of the elements in the table, if not stated otherwise):

```
[·]ᴸ(table[Products],2)                                  (Q₆)
  # ⇝ {:id=>7, :name=>p₂, :price=>20}
```

In rule (LL-LPOS), which captures positional access to a list, we link the positions in question (query q_{int} from step ②) with the list elements (query q_{list} from step ①) to obtain the proper elements by means of a comparison.

For this purpose, in step ③, each list element is temporarily extended with its absolute position and paired with q_{int}, so that we may qualify tuples by an equality comparison on the columns $\mathsf{item}_{|cs|+1}$ and $\mathsf{item}_{|cs|+2}$—step ④ reflects this situation.

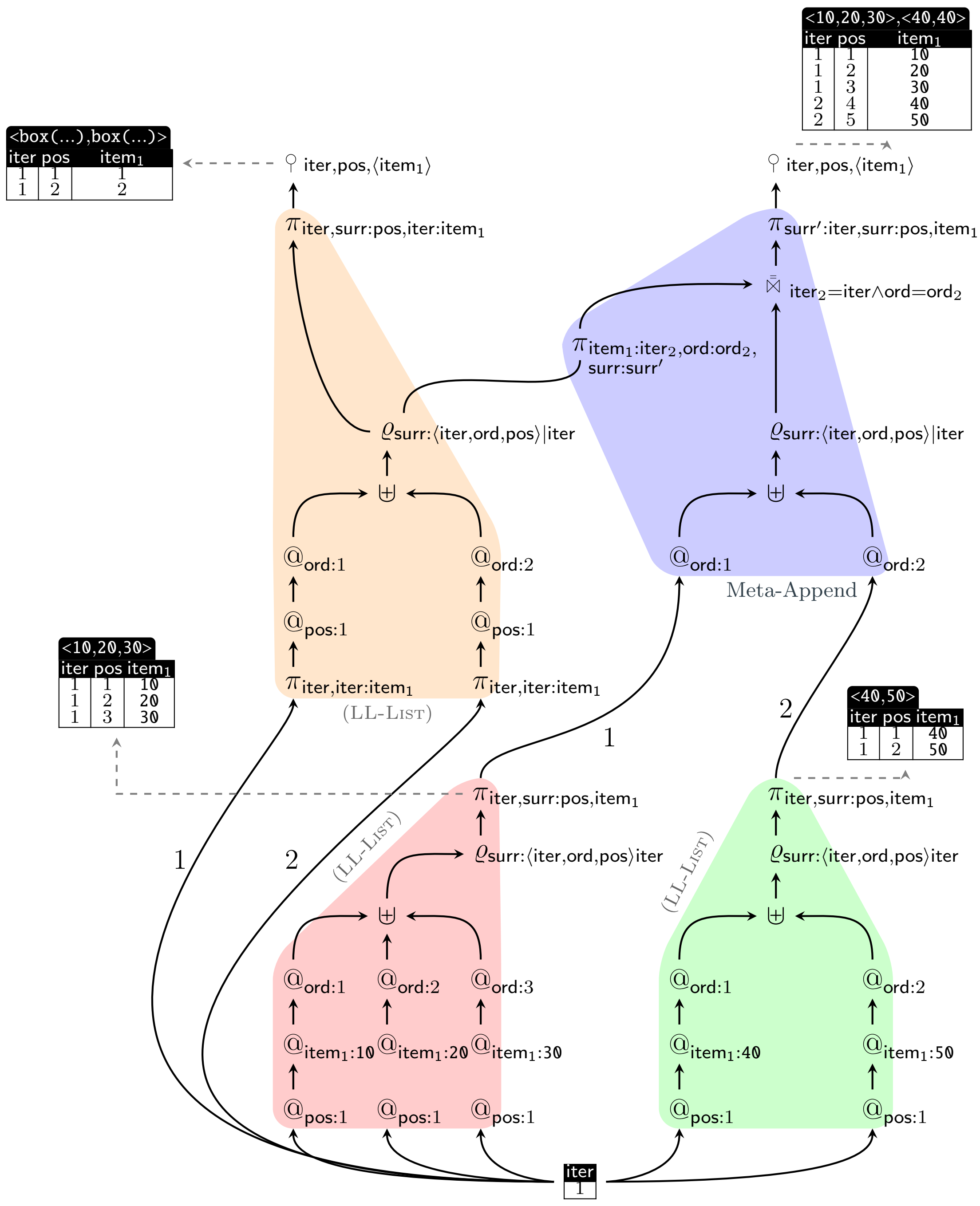


Figure 3.8: Loop lifted query plan resulting from the compilation of Query Q_3. The marked plan edges indicate where boxing splits the plans to calculate surrogate values (1 for `box(<10, 20, 30>)` and 2 for `box(<30, 40>)`). The inner lists are then merged into a single query (marked by the blue box).

$$\dfrac{\begin{array}{l} \textcircled{1}\ \Gamma; loop \vdash e_{\text{list}} \Rrightarrow (q_{\text{list}}, cs, surr_{\text{list}}) \\ \textcircled{2}\ \Gamma; loop \vdash e_{\text{int}} \Rrightarrow (q_{\text{int}}, \mathsf{item}_1, \emptyset) \\ \textcircled{3}\ q \equiv \curlyvee (\varrho_{\mathsf{item}_{|cs|+1}:\langle\mathsf{iter},\mathsf{pos}\rangle|\mathsf{iter}}(q_{\text{list}})^{|cs|+1}, q^1_{\text{int}}) \\ \textcircled{4}\ q_{\text{elem}} \equiv \pi_{\mathsf{iter},\mathsf{pos},\text{items}(cs)}(\sigma_{\mathsf{item}}(\circleddash_{\mathsf{item}:\left\langle\mathsf{item}_{|cs|+1},\mathsf{item}_{|cs|+2}\right\rangle}(q))) \end{array}}{\Gamma; loop \vdash \texttt{[}\cdot\texttt{]}^{\mathsf{L}}\texttt{(}e_{\text{list}}\texttt{,}e_{\text{int}}\texttt{)} \Rrightarrow (q_{\text{elem}}, cs, \llbracket surr_{\text{list}} \rrbracket^{\curlywedge}_{q_{\text{elem}}})} \text{(LL-LPos)}$$

3.13.2 The First and Last Element

The expression

```
first^L(table[Products])
  # ⇝ {:id=>4, :name=>p1, :price=>10}
```
(Q_7)

accesses the first element of a list and has the type $\texttt{first}^{\mathsf{L}} :: [\tau] \rightarrow \tau$. The close resemblance to positional access becomes clear when you take a look at rule (LL-LFirst) that is simply rewritten in terms of positional access.

$$\dfrac{\Gamma; loop \vdash \texttt{[}\cdot\texttt{]}^{\mathsf{L}}\texttt{(}e_{\text{list}}\texttt{,int[1])} \Rrightarrow (q_{\text{elem}}, cs_{\text{elem}}, surr_{\text{elem}})}{\Gamma; loop \vdash \texttt{first}^{\mathsf{L}}\texttt{(}e_{\text{list}}\texttt{)} \Rrightarrow (q_{\text{elem}}, cs_{\text{elem}}, surr_{\text{elem}})} \text{(LL-LFirst)}$$

Rule (LL-LLast) capturing the function $\texttt{last}^{\mathsf{L}} :: [\tau] \rightarrow \tau$ performs the task diametrically opposed to the above function by accessing the last element of a list. As in the above rule, the expression of the term is based on positional access and the function $\texttt{length}^{\mathsf{L}}\texttt{()}$, which derives the number of elements of a given list (we provide the rule in Section 3.14.1).

$$\dfrac{\Gamma; loop \vdash \texttt{[}\cdot\texttt{]}^{\mathsf{L}}\texttt{(}e_{\text{list}}\texttt{,length}^{\mathsf{L}}\texttt{(}e_{\text{list}}\texttt{))} \Rrightarrow (q_{\text{elem}}, cs_{\text{elem}}, surr_{\text{elem}})}{\Gamma; loop \vdash \texttt{last}^{\mathsf{L}}\texttt{(}e_{\text{list}}\texttt{)} \Rrightarrow (q_{\text{elem}}, cs_{\text{elem}}, surr_{\text{elem}})} \text{(LL-LLast)}$$

3.13.3 Prefix and Suffix

Rule (LL-LTake) and (LL-LDrop) render the functions that calculate a prefix or suffix of a list into their loop-lifted encoding. The function $\texttt{take}^{\mathsf{L}} :: [\tau] \times Int \rightarrow [\tau]$ expects a list (e_{list}) and an integer value (e_{int}) as arguments and returns a sublist consisting of the first e_{int} elements from the list e_{list}.

Poured into abstract syntax, this leads to a similar query as the following one that picks the initial three products (order is determined by the primary key):

```
take^L(table[Products],3)
  # ⇝ <{:id=>4, :name=>p1, :price=>10},
  #     {:id=>7, :name=>p2, :price=>20},
  #     {:id=>9, :name=>p3, :price=>30}>
```
(Q_8)

In rule (LL-LTAKE), we start with the translation of the list e_{list} in step ①. Then, we employ query q_{int} obtained by the translation of e_{int} (step ②) to only fetch the initial elements of the list. For this purpose we calculate in step ③ the absolute positions (column $\mathsf{item}_{|cs|+1}$) of the elements in each list based on the iteration identifier and the rank in column pos. The resulting query is paired with query q_{int} that provides the number of elements that are carried into the result for each iteration (renamed to column $\mathsf{item}_{|cs|+2}$ prevent name clashes).

In step ④ the absolute positions in each row are compared to column $\mathsf{item}_{|cs|+2}$ to filter all elements with a position that is smaller or equal. All elements whose positions meet this criterion pass the filter and end up in the final result. In step ⑤ we propagate new iteration context that has been established by the possible removal of elements to all surrogates.

$$\frac{\begin{array}{l} ①\ \Gamma; \mathit{loop} \vdash e_{\text{list}} \Rrightarrow (q_{\text{list}}, cs, \mathit{surr}_{\text{list}}) \\ ②\ \Gamma; \mathit{loop} \vdash e_{\text{int}} \Rrightarrow (q_{\text{int}}, \mathsf{item}_1, \emptyset) \\ ③\ q \equiv \curlyvee \left(\varrho_{\mathsf{item}_{|cs|+1}:\langle \mathsf{iter},\mathsf{pos}\rangle | \mathsf{iter}}(q_{\text{list}})^{|cs|+1}, q_{\text{int}}^{1}\right) \\ ④\ q' \equiv \pi_{\mathsf{iter},\mathsf{pos},\text{items}(cs)}\left(\sigma_{\mathsf{item}}\left(\circledless_{\mathsf{item}:\left\langle \mathsf{item}_{|cs|+1},\mathsf{item}_{|cs|+2}\right\rangle}(q)\right)\right) \end{array}}{⑤\ \Gamma; \mathit{loop} \vdash \mathtt{take}^{\mathsf{L}}\mathtt{(}e_{\text{list}}\mathtt{,}e_{\text{int}}\mathtt{)} \Rrightarrow (q', cs, \llbracket \mathit{surr}_{\text{list}} \rrbracket_{q}^{\curlywedge})} \text{(LL-LTAKE)}$$

The function $\mathtt{drop}^{\mathsf{L}} :: [\tau] \times \mathit{Int} \to [\tau]$ discards the initial e_{int} elements from the list e_{list} (with e_{list} as the first argument and e_{int} as the second argument). Rule (LL-LDROP) reprises the steps from rule (LL-LTAKE) with the exception that the elements in each list whose absolute position is smaller or equal do not reappear in the final result.

$$\frac{\begin{array}{l} \Gamma; \mathit{loop} \vdash e_{\text{list}} \Rrightarrow (q_{\text{list}}, cs, \mathit{surr}_{\text{list}}) \\ \Gamma; \mathit{loop} \vdash e_{\text{int}} \Rrightarrow (q_{\text{int}}, \mathsf{item}_1, \emptyset) \\ q \equiv \curlyvee \left(\varrho_{\mathsf{item}_{|cs|+1}:\langle \mathsf{iter},\mathsf{pos}\rangle | \mathsf{iter}}(q_{\text{list}})^{|cs|+1}, q_{\text{int}}^{1}\right) \\ q' \equiv \pi_{\mathsf{iter},\mathsf{pos},\text{items}(cs)}\left(\sigma_{\mathsf{item}}\left(\circledgtr_{\mathsf{item}:\left\langle \mathsf{item}_{|cs|+1},\mathsf{item}_{|cs|+2}\right\rangle}(q)\right)\right) \end{array}}{\Gamma; \mathit{loop} \vdash \mathtt{drop}^{\mathsf{L}}\mathtt{(}e_{\text{list}}\mathtt{,}e_{\text{int}}\mathtt{)} \Rrightarrow (q', cs, \llbracket \mathit{surr}_{\text{list}} \rrbracket_{q}^{\curlywedge})} \text{(LL-LDROP)}$$

3.14 More List Functions

For the following simple list functions, we provide a minimal example that illustrates their semantics along with the inference rule that prepares the loop-lifted encoding.

3.14.1 Length

The length of a list is calculated by the function $\texttt{length}^{\mathsf{L}} :: [\tau] \rightarrow \mathit{Int}$. The following query determines the length of a nested list. Note that the length is not affected by the inner lists.

```
lengthᴸ(<<10,20,30>,<40,50>>) # ⇝ 2   (Q₉)
```

Rule (LL-Length) captures this behavior by breaking down the loop-lifted representation of the result derived by query q_{list} into several partitions according to the list identifier described by the column iter. In turn, the aggregate function COUNT is eventually employed to count the elements for each list.

$$\frac{\begin{array}{c}\Gamma; \mathit{loop} \vdash e_{\text{list}} \Rrightarrow (q_{\text{list}}, cs_{\text{list}}, surr_{\text{list}}) \\ q \equiv @_{pos:1}(\mathrm{GRP}_{\mathsf{item_1}:\mathrm{COUNT}(\mathsf{item_1})/\mathsf{iter}}(q))\end{array}}{\Gamma; \mathit{loop} \vdash \texttt{length}^{\mathsf{L}}\texttt{(}e_{\text{list}}\texttt{)} \Rrightarrow (q_{\text{length}}, \mathsf{item_1}, \emptyset)} \quad \text{(LL-Length)}$$

3.14.2 Flatten

The function $\texttt{flatten} :: [[\tau]] \rightarrow [\tau]$ disregards the outermost nesting level from its input list:

```
flatten(<<10,20,30>,<40,50>>) # ⇝ <10,20,30,40>   (Q₁₀)
```

Observe that the function `flatten()` is operationally equivalent to `unbox()`, which merges a nested expression into a single query by means of a equi-join on the surrogate columns. Rule (LL-Flatten) reflects this behavior.

$$\frac{\Gamma; \mathit{loop} \vdash \texttt{unbox(}e_{\text{list}}\texttt{)} \Rrightarrow pb}{\Gamma; \mathit{loop} \vdash \texttt{flatten(}e_{\text{list}}\texttt{)} \Rrightarrow pb} \quad \text{(LL-Flatten)}$$

3.14.3 Uniq

The list function $\texttt{uniq} :: [\,\mathrm{atom}(\tau)] \rightarrow [\,\mathrm{atom}(\tau)]$ removes all duplicates from its argument (e_{list}). To successfully complete its task, the elements of the input list (e_{list}) must occupy exactly a single row because the implementing query relies on the duplicate-elimination operator (δ). The following query illustrates how the function eliminates the value `20` that occurs twice in the given list. Note that the arrangement of the elements in the resulting list is arbitrary and cannot be properly restored with respect to possibly removed elements.

```
uniq(<10,20,30,20>) # ⇝ <20,10,30>   (Q₁₁)
```

In rule (LL-UNIQ) the duplicate-elimination operator only eliminates tuples originating in the same iteration. Because the order of elements cannot be restored, we set up the position column with 1 (the choice for a concrete position is arbitrary, however) to reflect this situation.

$$\frac{\Gamma; \mathit{loop} \vdash e_{\text{list}} \Rrightarrow (q_{\text{list}}, cs_{\text{list}}, \emptyset) \qquad q \equiv @_{\mathsf{pos}:1}(\delta(\pi_{\mathsf{iter},\text{items}(cs_{\text{list}})}(q_{\text{list}})))}{\Gamma; \mathit{loop} \vdash \texttt{uniq(}e_{\text{list}}\texttt{)} \Rrightarrow (q, cs_{\text{list}}, \emptyset)} \text{(LL-UNIQ)}$$

3.14.4 Reducing Lists

SWITCH provides several *aggregate functions* on lists, performing operations that reduces a list to a single value that that captures some property of the list. A popular function is **sum** $::$ [num(τ)] $\rightarrow$ num τ that calculates the total of all values in the argument list. With regard to this function we exemplify the translation of aggregate functions. The remaining aggregate functions deriving the average, minimum and maximum on a list can be found in rule (LL-AVG), rule (LL-MIN) and rule (LL-MAX), respectively.

Applied to a list of numeric values the function calculates the total, as in the below query:

```
sum(<10,20,30,20>) # ⇝ 80
```
(Q_{12})

The result of translation of list e_{list} in rule (LL-SUM) is expected to be the loop-lifted encoding of a list containing numeric values. Since the column iter assumes the role of a list identifier each list describes a group of rows based on an equal value in column iter within query q_{list}. We consider each group separately by breaking the result of query q_{list} down into partitions with equal list identifiers and apply the proper aggregate function SUM on the values residing in the column item_1. As a result we obtain a single row for each list containing the iteration column side by side with the overall sum. We conclude the rule by restoring the position column that has been lost in the course of aggregation.

$$\frac{\Gamma; \mathit{loop} \vdash e_{\text{list}} \Rrightarrow (q_{\text{list}}, \mathsf{item}_1, \emptyset) \qquad q \equiv @_{pos:1}(\text{GRP}_{\mathsf{item}_1:\text{SUM}(\mathsf{item}_1)/\mathsf{iter}}(q_{\text{list}}))}{\Gamma; \mathit{loop} \vdash \texttt{sum(}e_{\text{list}}\texttt{)} \Rrightarrow (q, \mathsf{item}_1, \emptyset)} \text{(LL-SUM)}$$

$$\frac{\Gamma; \mathit{loop} \vdash e_{\text{list}} \Rrightarrow (q_{\text{list}}, \mathsf{item}_1, \emptyset) \qquad q \equiv @_{pos:1}(\text{GRP}_{\mathsf{item}_1:\text{AVG}(\mathsf{item}_1)/\mathsf{iter}}(q_{\text{list}}))}{\Gamma; \mathit{loop} \vdash \texttt{avg(}e_{\text{list}}\texttt{)} \Rrightarrow (q, \mathsf{item}_1, \emptyset)} \text{(LL-AVG)}$$

$$\frac{\Gamma; \mathit{loop} \vdash e_{\text{list}} \Rrightarrow (q_{\text{list}}, \mathsf{item}_1, \emptyset) \qquad q \equiv @_{\mathit{pos}:1}(\mathrm{GRP}_{\mathsf{item}_1:\textsc{min}(\mathsf{item}_1)/\mathsf{iter}}(q_{\text{list}}))}{\Gamma; \mathit{loop} \vdash \texttt{min(}e_{\text{list}}\texttt{)} \Rrightarrow (q, \mathsf{item}_1, \emptyset)} \text{(LL-MIN)}$$

$$\frac{\Gamma; \mathit{loop} \vdash e_{\text{list}} \Rrightarrow (q_{\text{list}}, \mathsf{item}_1, \emptyset) \qquad q \equiv @_{\mathit{pos}:1}(\mathrm{GRP}_{\mathsf{item}_1:\textsc{max}(\mathsf{item}_1)/\mathsf{iter}}(q_{\text{list}}))}{\Gamma; \mathit{loop} \vdash \texttt{max(}e_{\text{list}}\texttt{)} \Rrightarrow (q, \mathsf{item}_1, \emptyset)} \text{(LL-MAX)}$$

3.15 Iteration

The higher order function `map` :: $[\tau] \times (\tau \to \sigma) \to [\sigma]$ passes the elements of the input list (first argument) successively to the user-supplied function (second argument). In turn, the function is evaluated for each element to provide the sub-results assembled to form the resulting list. The order in which the sub-results are assembled is prescribed by the logical order of the elements in the input list. All following iteration-based constructs use this prototypical iteration to build upon their functionality.

$$\frac{\begin{array}{l} ① \ \Gamma; \mathit{loop} \vdash e_{\text{list}} \Rrightarrow (q_{\text{list}}, \mathit{cs}_{\text{list}}, \mathit{surr}_{\text{list}}) \\ ② \ q_{\text{inner}} \equiv \vec{\varrho}_{\mathsf{inner}:\langle\mathsf{iter},\mathsf{pos}\rangle}(q_{\text{list}}) \\ ③ \ \mathit{map} \equiv \pi_{\mathsf{iter:outer,inner,pos:pos_{\text{outer}}}}(q_{\text{inner}}) \\ ④ \ q_x \equiv @_{\mathsf{pos}:1}(\pi_{\mathsf{inner:iter},\mathrm{items}(\mathit{cs})}(q_{\text{inner}})) \\ ⑤ \ \Gamma_x \equiv (\!|\Gamma|\!)^{\uparrow}_{\mathit{map}} \,\dot\cup\, \{x \mapsto (q_x, \mathit{cs}_{\text{list}}, \mathit{surr}_{\text{list}})\} \\ ⑥ \ \mathit{loop}_x = \pi_{\mathsf{inner:iter}}(\mathit{map}) \\ ⑦ \ \Gamma_x; \mathit{loop}_x \vdash e_{\text{body}} \Rrightarrow (q_{\text{body}}, \mathit{cs}_{\text{body}}, \mathit{surr}_{\text{body}}) \\ ⑧ \ q_{\text{map}} \equiv \pi_{\mathsf{outer:iter,pos_{\text{outer}}:pos},\mathrm{items}(\mathit{cs}_{\text{body}})}(\mathit{map} \bowtie_{\mathsf{inner=iter}} q_{\text{body}}) \end{array}}{\Gamma; \mathit{loop} \vdash \texttt{map(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{body}}\texttt{)} \Rrightarrow (q_{\text{map}}, \mathit{cs}_{\text{body}}, \mathit{surr}_{\text{body}})} \text{(LL-MAP)}$$

The single inference rule (LL-MAP) summarizes several steps to express the function `map()` by means of tabular operators. In step ① we compile the list e_{list} in the current iteration scope to its loop-lifted encoding and employ the resulting query (q_{list}) to establish a new iteration context for the function body (hosted by the column inner in step ②).

Based on the query q_{inner} in step ② we derive the query *map* that relates the outer and inner iteration context: for every inner iteration *map* preserves the iteration identifiers of the parent scope. To further indicate this intention we rename the column iter to outer in step ③. Note that we likewise preserve the position that reflects the logical order among elements in query q_{list} (temporarily renamed to $\mathsf{pos}_{\text{outer}}$) to prevent its costly recalculation.

Query *map* is then employed in step ⑤ to lift the tabular representation for every variable in the variable environment Γ into the new iteration context hosted by environment Γ_x, so that they may participate in the evaluation of e_{body}.

In step ④ we prepare the loop-lifted representation of query q_x to describe the loop variable x, which in turn is inserted into the variable environment. In this new iteration context and variable environment we fork the translation of the function body (e_{body}) (step ⑦) to derive all sub-results.

To conclude the rule we assemble the sub-results back from the independent iterations into a single list. Query *map* provides the necessary connection between the enclosing iteration scopes as well as the prescribed order, so that we may obtain the iteration identifiers from the enclosing scope by means of an equi-join in step ⑧.

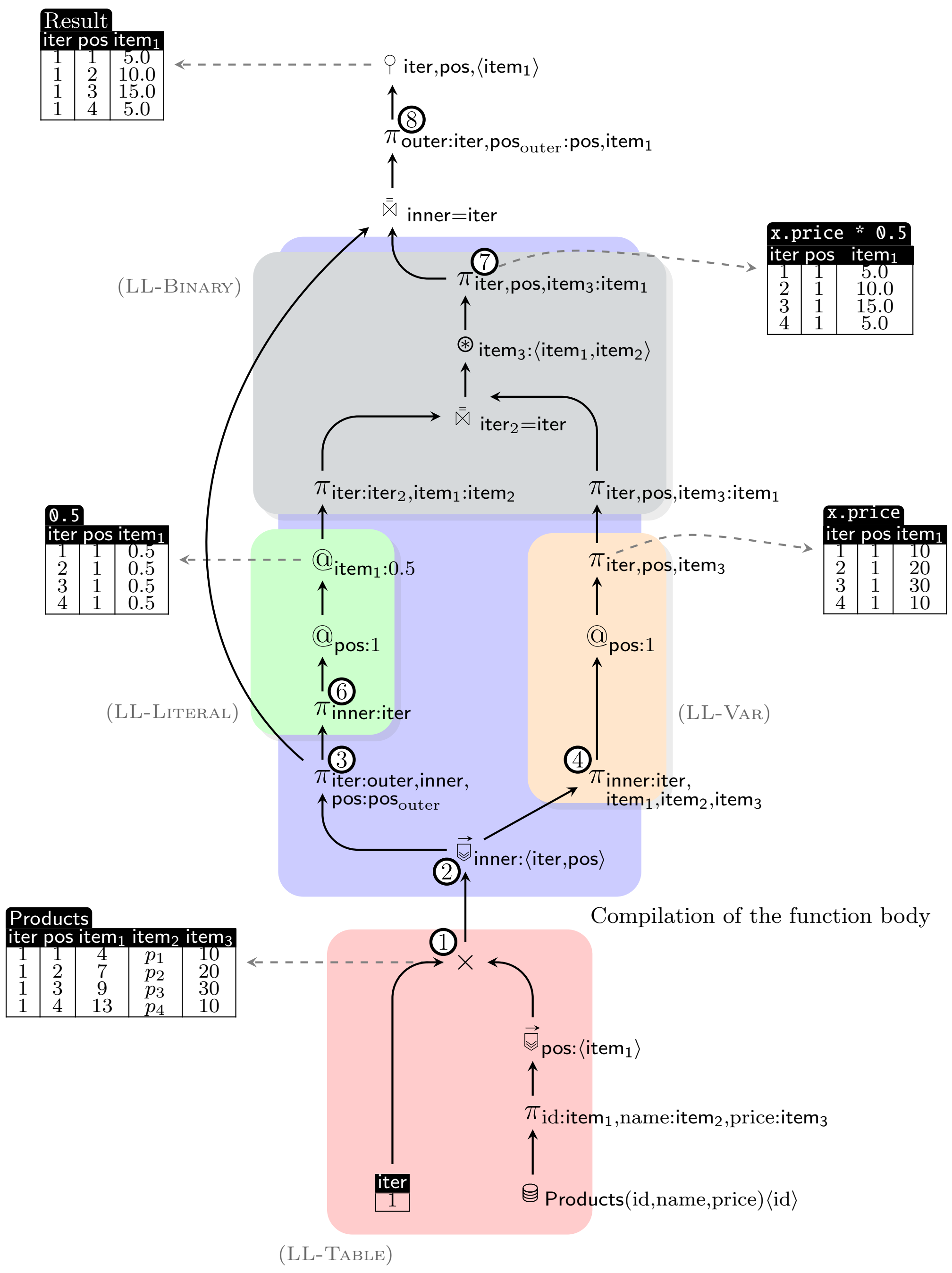
For an illustration consider the below SWITCH snippet, which computes a 50 percent discount for the price of each product in the table Products.

```
Products.map {|p| p.price * 0.5}
                 ≡
map(table[Products],λp.*(proj(var[p],price),0.5))      (Q13)
  # ⇝ <5,10,15,5>
```

The query plan in Figure 3.9 shows the resulting tabular representation where several rules elegantly interact to present the overall result. In a nutshell, the database back-end poses the schema of table Products to trigger the rule (LL-TABLE). The resulting query is consequently used to set up new the iteration context to compile the function body. Within the compilation of the function body, the rules (LL-LITERAL), (LL-VAR), and (LL-BINARY) participate to derive the discount on all product prices. With the sub-results at hand, the overall result is assembled by means of an equi-join on query *map*, which establishes the outer context.

The function `flat_map` $:: [\tau] \times (\tau \to [\sigma]) \to [\sigma]$ takes a list as its first argument and a user-supplied function as the second argument. The latter is obliged to evaluate to a list for each element in the input list. The resulting *list of lists* is then flattened. In rule (LL-FLATMAP) we show how this macro expands into a composition of the functions `map()` and `flatten()`.

$$\frac{\Gamma; \mathit{loop} \vdash \texttt{flatten(map(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{body}}\texttt{))} \Rrightarrow pb}{\Gamma; \mathit{loop} \vdash \texttt{flat\_map(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{body}}\texttt{)} \Rrightarrow pb} \text{ (LL-FLATMAP)}$$



ϱ iter,pos,⟨item$_1$⟩
8
$\pi_{\text{outer:iter,pos}_{\text{outer}}\text{:pos,item}_1}$
$\bar{\bowtie}$ inner=iter
7
$\pi_{\text{iter,pos,item}_3\text{:item}_1}$
$\circledast$ item$_3$:⟨item$_1$,item$_2$⟩
$\bar{\bowtie}$ iter$_2$=iter
$\pi_{\text{iter:iter}_2\text{,item}_1\text{:item}_2}$
$\pi_{\text{iter,pos,item}_3\text{:item}_1}$
(LL-Binary)
@item$_1$:0.5
@pos:1
6
$\pi_{\text{inner:iter}}$
(LL-Literal)
$\pi_{\text{iter,pos,item}_3}$
@pos:1
(LL-Var)
3
π iter:outer,inner, pos:pos$_{\text{outer}}$
4
π inner:iter, item$_1$,item$_2$,item$_3$
2
inner:⟨iter,pos⟩
Compilation of the function body
1
$\times$
pos:⟨item$_1$⟩
$\pi_{\text{id:item}_1\text{,name:item}_2\text{,price:item}_3}$
Products(id,name,price)⟨id⟩
(LL-Table)

Result

iter	pos	item$_1$
1	1	5.0
1	2	10.0
1	3	15.0
1	4	5.0

x.price * 0.5

iter	pos	item$_1$
1	1	5.0
2	1	10.0
3	1	15.0
4	1	5.0

0.5

iter	pos	item$_1$
1	1	0.5
2	1	0.5
3	1	0.5
4	1	0.5

x.price

iter	pos	item$_1$
1	1	10
2	1	20
3	1	30
4	1	10

Products

iter	pos	item$_1$	item$_2$	item$_3$
1	1	4	p_1	10
1	2	7	p_2	20
1	3	9	p_3	30
1	4	13	p_4	10

iter
1

Figure 3.9: Query plan resulting from the compilation of Query Q_{13}. The plan annotations exemplify the different steps summarized by rule (LL-MAP), while the tables on the left and on the right side tap the intermediate results at the respective tabular operator during query execution.

3.15.1 Filtering Elements

Similar to `map()` the higher order function `select` $:: [\tau] \times (\tau \to \mathit{Bool}) \to [\tau]$ iterates over all elements in the input list to evaluate a user-supplied predicate for each element This predicate decides whether an element eventually ends up in the result list.

The following query selects all products from the table Products that are cheaper than `20`; the products passing the filter are presented in a list:

```
Products.select {|p| p.price < 20}
```

$$\equiv$$

$$\texttt{select(table[Products],}\lambda p.\texttt{<(proj(var[}p\texttt{],price),20))} \quad (Q_{14})$$

```
# ⇝ <{:id=>4, :name=>p₁, :price=>10},
#     {:id=>13, :name=>p₄, :price=>10}>
```

With a slight extension, we may seamlessly turn `map()` into a filter, dropping elements of the binding list that do not pass the filter criteria. Rule (LL-Select) summarizes the steps, we will sketch below, in a single inference rule:

In step ①, in a single iteration we preserve each element of the input list (e_{list}) along with the evaluation of the body of the predicate (e_{pred}), which is expected to evaluate to a boolean value hosted by column $\mathsf{item}_{|cs|+1}$.

In step ②, we drop all elements with the predicate evaluating to `false`. The resulting iteration context from query q' is then established in the surrogate maps in step ③ to account (possibly) nested lists. This concluding step eliminates the leftover elements that originate in an element dropped in step ②.

$$\frac{\begin{array}{l}① \ \Gamma; \mathit{loop} \vdash \texttt{map(}e_{\text{list}}\texttt{,}\lambda x.\texttt{(}x, e_{\text{pred}}\texttt{))} \Rrightarrow (q, (cs_x, \mathsf{item}_{|cs_x|+1}), \mathit{surr}) \\ ② \ q' \equiv \pi_{\mathsf{iter,pos,}\mathrm{items}(cs_x)}(\sigma_{\mathsf{item}_{|cs_x|+1}}(q))\end{array}}{③ \ \Gamma; \mathit{loop} \vdash \texttt{select(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{pred}}\texttt{)} \Rrightarrow (q', cs_x, \llbracket \mathit{surr} \rrbracket^{\lambda}_{q'})} \ \text{(LL-Select)}$$

To adhere to the semantics of function $\texttt{reject(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{pred}}\texttt{)}$ we simply invert the user-supplied predicate (e_{pred}) to let the loop-lifted representation reject any item that meets the criteria. The rule (LL-Reject) implements this behavior.

$$\frac{\Gamma; \mathit{loop} \vdash \texttt{select(}e_{\text{list}}\texttt{,}\lambda x.\texttt{not(}e_{\text{pred}}\texttt{))} \Rrightarrow pb}{\Gamma; \mathit{loop} \vdash \texttt{reject(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{pred}}\texttt{)} \Rrightarrow pb} \ \text{(LL-Reject)}$$

The function `count` $:: [\tau] \times (\tau \to \mathit{Bool}) \to \mathit{Int}$ exhibits a similar semantics. Applied to a list it counts the elements that meet a user-supplied condition, as exemplified in the following query:

```
Products.count {|p| p.price = 10}
                   ≡
count(table[Products],λp.=(proj(var[p],price),10))      (Q15)
  # ⇝ 2
```

By dropping the elements that do not fulfill the user-supplied criteria from the input list and counting the remaining elements, rule (LL-COUNT) composes two operations and results in the following simple rewrite.

$$\frac{\Gamma; \mathit{loop} \vdash \texttt{length}^{\mathsf{L}}\texttt{(select(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{pred}}\texttt{))} \Rrightarrow (q, \mathsf{item}_1, \emptyset)}{\Gamma; \mathit{loop} \vdash \texttt{count(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{pred}}\texttt{)} \Rrightarrow (q, \mathsf{item}_1, \emptyset)} \quad \text{(LL-COUNT)}$$

3.15.2 Establishing Order

In function `sort_by` $:: [\tau] \times (\tau \rightarrow \text{atom}(\sigma)) \rightarrow [\tau]$ the user-supplied function (second argument) is used to establish an arrangement of all elements in the input list (first argument). The result of the independent evaluation of the function for all elements in the input list determines the new order. If the function evaluates to a tuple or record that may even be nested, as long as they do not contain lists. The elements in the input list are arranged in ascending order according to the lexicographic order of their underlying atomic values.

To exemplify how this affects the logical order of elements, consider the below query, in which the price prescribes the arrangement of elements in the resulting list. Note that due to the equal price, the first two elements may swap positions in the result:

```
Products.sort_by {|p| p.price }
                   ≡
sort_by(table[Products],λp.proj(var[p],price))
  # ⇝ <{:id=> 4, :name=>p1, :price=>10},   (Q16)
  #     {:id=>13, :name=>p4, :price=>10},
  #     {:id=> 7, :name=>p2, :price=>20},
  #     {:id=> 9, :name=>p3, :price=>30}>
```

We build sorting on top of `map()` to derive the sorting criterion while preserving the elements of the binding list. Based on the sorting criterion described by the columns items(cs_{ord}), a new position rank ($\mathsf{pos}_{\text{new}}$) is then established in order to rearrange the elements.

$$\frac{\begin{array}{c}\Gamma; \mathit{loop} \vdash \texttt{map(}e_{\text{list}}\texttt{,}\lambda x.\texttt{(}x, e_{\text{ord}}\texttt{))} \Rrightarrow (q, (cs_x, cs_{\text{ord}}), \mathit{surr}) \\ q' \equiv \pi_{\mathsf{iter},\mathsf{pos}_{\text{new}}:\mathsf{pos},\text{items}(cs_x)}(\overrightarrow{\varrho}_{\mathsf{pos}_{\text{new}}:\langle\text{items}(cs_{\text{ord}})\rangle}(q))\end{array}}{\Gamma; \mathit{loop} \vdash \texttt{sort\_by(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{ord}}\texttt{)} \Rrightarrow (q', cs_x, \mathit{surr})} \quad \text{(LL-SORTBY)}$$

As expressed in rule (LL-LREVERSE), to revert the elements of a list we derive a new position column ($\mathsf{pos}_{\text{new}}$) based on the descending order of the current position column (pos). This operation rearranges the ranks in the current position column in the opposite direction.

$$\frac{\Gamma; \mathit{loop} \vdash e_{\text{list}} \Rrightarrow (q, \mathit{cs}, \mathit{surr}) \qquad q' \equiv \pi_{\mathsf{iter},\mathsf{pos}_{\text{new}}:\mathsf{pos},\text{items}(\mathit{cs})}(\vec{\Downarrow}_{\mathsf{pos}_{\text{new}}:\langle \mathsf{pos}:\mathit{desc}\rangle}(q))}{\Gamma; \mathit{loop} \vdash \texttt{reverse}^{\mathsf{L}}(e_{\text{list}}) \Rrightarrow (q', \mathit{cs}, \mathit{surr})} \quad \text{(LL-LReverse)}$$

3.15.3 TakeWhile and DropWhile

The function `take_while` $:: [\tau] \times (\tau \to \mathit{Bool}) \to [\tau]$ derives the prefix of the input list according to the user-supplied predicate (second argument). The elements of the input list are *transfered* into the final result as long as the predicate holds, adhering to the semantics described in [FM08, ch. 9, p. 322ff].

For a minimal example consider the below query, which successively iterates through all products. The products reappear in the final result as long as they are cheaper than `30`. Note that even though p_4 satisfies this condition, it is not found in the resulting list (see Figure 3.5 on page 75 to examine the table Products).

```
          Products.take_while {|p| p.price < 30 }
                              ≡
take_while(table[Products],λp.<(proj(var[p],price),30))   (Q17)
  # ⇝ [{:id=>4, :name=>p1, :price=>10},
  #     {:id=>7, :name=>p2, :price=>20}]
```

In rule (LL-TAKEWHILE) this is implemented by rewriting `take_while()` by means of an iteration that evaluates the predicate for each element in the list e_{list} and at the same time retains the elements themselves (step ①). Essentially, in step ③ we invert the predicate to obtain the position of the first element that does not meet the criterion—this and all following elements do not appear in the final result. Query $q_{\text{pos}\cdot\text{min}}$, which contains this critical position, is paired with q_{list} that is now confined to all prior elements by means of a simple comparison on the position ranks (step ④).

A problem, however, arises in the formulation of query $q_{\text{pos}\cdot\text{min}}$: if all elements in e_{list} match the predicate, its list identifier disappears entirely from query $q_{\text{pos}\cdot\text{min}}$ and all constitutive queries. To prevent this situation, we place an artificial `false` value at the end of the list (step ②). This ensures that even though all elements satisfy the predicate, query $q_{\text{pos}\cdot\text{min}}$ passes a proper position rank.

$$\frac{\begin{array}{l}① \ \Gamma; \mathit{loop} \vdash \mathtt{map(}e_{\text{list}}\mathtt{,}\lambda x.\mathtt{(}x, e_{\text{pred}}\mathtt{))} \Rrightarrow (q_{\text{list}}, (cs_x, \mathsf{item}_{|cs_x|+1}), \mathit{surr}) \\ ② \left\{\begin{array}{l} q_{\text{pred}} \equiv \pi_{\mathsf{iter,pos,item}_{|cs_x|+1}:\mathsf{item}_1}(q_{\text{list}}) \\ q_{\text{pos}\cdot\text{max}} \equiv \text{GRP}_{\mathsf{item}_1:\text{MAX}(\mathsf{pos})/\mathsf{iter}}(q_{\text{pred}}) \\ q_{\text{artif}} \equiv @_{\mathsf{item}_1:\mathtt{false}}(\pi_{\mathsf{iter,pos}}(\oplus_{\mathsf{pos}:\langle \mathsf{item}_1,\mathsf{item}_2\rangle}(@_{\mathsf{item}_2:1}(q_{\text{pos}\cdot\text{max}})))) \\ q'_{\text{pred}} \equiv q_{\text{pred}} \uplus q_{\text{artif}} \end{array}\right. \\ ③ \ q_{\text{pos}\cdot\text{min}} \equiv \text{GRP}_{\mathsf{item}_1:\text{MIN}(\mathsf{pos})/\mathsf{iter}}(\sigma_{\mathsf{pass}}(\ominus_{\mathsf{pass}:\langle \mathsf{item}_1\rangle}(q'_{\text{pred}}))) \\ ④ \ q \equiv \pi_{\mathsf{iter,pos,}\text{items}(cs_x)}(\sigma_{\mathsf{pass}}(\otimes_{\mathsf{pass}:\langle \mathsf{pos,item}_{|cs_x|+2}\rangle}(\curlyvee\ (q_{\text{list}}^{|cs_x|+1}, q_{\text{pos}\cdot\text{min}}^{1}))))\end{array}}{\Gamma; \mathit{loop} \vdash \mathtt{take\_while(}e_{\text{list}}\mathtt{,}\lambda x.e_{\text{pred}}\mathtt{)} \Rrightarrow (q, cs_x, \llbracket \mathit{surr} \rrbracket_q^{\curlywedge})} \quad \text{(LL-TakeWhile)}$$

As opposed to function **take_while()** the function **drop_while** :: $[\tau] \times (\tau \to \mathit{Bool}) \to [\tau]$ successively *removes* elements from the input list as long as the predicate holds; all following elements reappear in the final result. Again, the predicate is inverted in order to obtain the position of the first element in the list that violates the condition. Based on this position, all following elements are carried into the end result. Note that we must not extend the list with an artificial **false** value: if all elements match the predicate in order to be removed, the list identifier disappears likewise in query $q_{\text{pos}\cdot\text{min}}$. In this case, however, this behavior adheres to the semantics of **drop_while()** [FM08, ch. 9, p. 322ff] that returns an empty list.

$$\frac{\begin{array}{l}\Gamma; \mathit{loop} \vdash \mathtt{map(}e_{\text{list}}\mathtt{,}\lambda x.\mathtt{(}x, e_{\text{pred}}\mathtt{))} \Rrightarrow (q_{\text{list}}, (cs_x, \mathsf{item}_{|cs_x|+1}), \mathit{surr}) \\ q_{\text{pos}\cdot\text{min}} \equiv \text{GRP}_{\mathsf{item}_1:\text{MIN}(\mathsf{pos})/\mathsf{iter}}(\sigma_{\mathsf{pass}}(\ominus_{\mathsf{pass}:\langle \mathsf{item}_{|cs_x|+1}\rangle}(q_{\text{list}}))) \\ q \equiv \pi_{\mathsf{iter,pos,}\text{items}(cs_x)}(\sigma_{\mathsf{pass}}(\otimes_{\mathsf{pass}:\langle \mathsf{pos,item}_{|cs_x|+2}\rangle}(\curlyvee\ (q_{\text{list}}^{|cs_x|+1}, q_{\text{pos}\cdot\text{min}}^{1}))))\end{array}}{\Gamma; \mathit{loop} \vdash \mathtt{drop\_while(}e_{\text{list}}\mathtt{,}\lambda x.e_{\text{pred}}\mathtt{)} \Rrightarrow (q, cs_x, \llbracket \mathit{surr} \rrbracket_q^{\curlywedge})} \quad \text{(LL-DropWhile)}$$

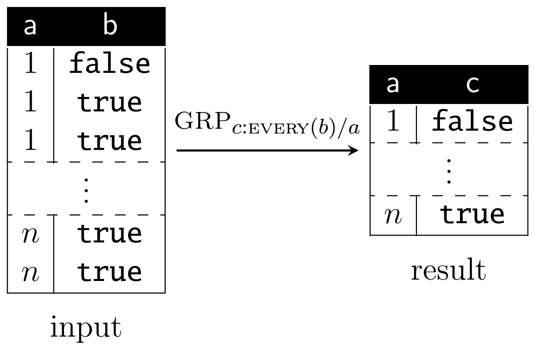
3.15.4 Quantification and Element Lookup

To check whether all elements in a list satisfy a certain condition consider the following query. In this case, the function **all?** :: $[\tau] \times (\tau \to \mathit{Bool}) \to \mathit{Bool}$ evaluates to **true** because all elements in the binding list are greater or equal **10**. For an empty binding list the function returns **true** for an arbitrary condition:

```
all?(<10,20,30,20>,λx.>=(var[x],10)) # ⇝ true   (Q18)
```

To calculate the loop-lifted encoding in rule (LL-ALL) we rely on the aggregate operator EVERY: For a boolean column b and the grouping column a, the operator $\text{GRP}_{c:\text{EVERY}(b)/a}(q)$ provides the aggregate conjunction on column b for each input group in column c (see Figure 3.10).

In step ①, after having evaluated the condition for each element in e_{list}, we employ the aforementioned operator to compute the aggregated conjunction on



a	b
1	false
1	true
1	true
⋮	
n	true
n	true

input

$\xrightarrow{\text{GRP}_{c:\text{EVERY}(b)/a}}$

a	c
1	false
⋮	
n	true

result

Figure 3.10: Demostrates the application of EVERY on the input table on left-hand side.

the resulting query (step ②). In step ② we account possible empty lists that, represented by the absence of a surrogate value in q, are present in the current iteration context ($loop$).

$$\frac{\begin{array}{l}① \ \Gamma; loop \vdash \texttt{map(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{pred}}\texttt{)} \Rrightarrow (q, \mathsf{item}_1, \emptyset) \\ ② \ q' \equiv \text{GRP}_{\mathsf{item}_1:\text{EVERY}(\mathsf{item}_1)/\mathsf{iter}}(q) \\ ③ \ q'' \equiv @_{\mathsf{pos}:1}(@_{\mathsf{item}_1:\texttt{true}}(\pi_{\mathsf{iter}}(loop) \ / \ q') \uplus q')\end{array}}{\Gamma; loop \vdash \texttt{all?(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{pred}}\texttt{)} \Rrightarrow (q'', \mathsf{item}_1, \emptyset)} \quad \text{(LL-ALL)}$$

For a demonstrating example of function **any?** $:: [\tau] \times (\tau \rightarrow Bool) \rightarrow Bool$ consider the below query, in which we check for the input list, if at least one element is greater or equal `30`. For an empty list the function evaluates to **false** for an arbitrary condition:

`any?(<10,20,30,20>,`λx`.>=(var[`x`],30))  # ⇝ true` (Q_{19})

In step ① we eliminate all elements from e_{list} that do not fulfill the condition, so that lists in which all elements fail to satisfy the condition do not occur in q. In step ② we retrieve the remaining lists, of which at least one element met the respective criteria. In a way similar to that of rule (LL-ALL), for the failing lists and empty lists we employ the current iteration context to compute the overall result (step ③).

$$\frac{\begin{array}{l}① \ \Gamma; loop \vdash \texttt{select(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{pred}}\texttt{)} \Rrightarrow (q, cs, surr) \\ ② \ q' \equiv @_{\mathsf{item}:\texttt{true}}(\delta(\pi_{\mathsf{iter}}(q))) \\ ③ \ q'' \equiv @_{\mathsf{pos}:1}(@_{\mathsf{item}_1:\texttt{false}}(\pi_{\mathsf{iter}}(loop) \ / \ q') \uplus q')\end{array}}{\Gamma; loop \vdash \texttt{any?(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{pred}}\texttt{)} \Rrightarrow (q'', \mathsf{item}_1, \emptyset)} \quad \text{(LL-ANY)}$$

For the following query, the function `member?` :: [atom(τ)] × atom(τ) → *Bool* succeeds because `30` exists in the input list:

```
member?(<10,20,30,20>,30) # ⇝ true   (Q_20)
```

It is important to observe that `member?()` is simply a special case of the function `all?()` if the condition is stated accordingly. This situation is expressed by rule (LL-MEMBER) in form of a simple rewrite.

$$\frac{\Gamma; loop \vdash \texttt{any?(}e_{\text{list}}\texttt{,}\lambda x.\texttt{=(var[}x\texttt{]},e\texttt{))} \Rrightarrow pb}{\Gamma; loop \vdash \texttt{member?(}e_{\text{list}}\texttt{,}e\texttt{)} \Rrightarrow pb}\ \text{(LL-MEMBER)}$$

3.15.5 Grouping

The function `group_with` :: [τ] × (τ → atom(σ)) → [[τ]] classifies the elements of the input list according to the user-supplied discriminator provided by the second argument. Elements that are tied according to the discriminator are placed into a single list in the second tuple component of the overall result; the value of the respective discriminator is preserved in the first tuple component.

In the below example the products (in table Products) are grouped accordingly to their price. The equally-priced products p_1 and p_4 land in the same group while the remaining values form a separate group.

```
    Products.group_with {|p| p.price }
                        ≡
group_with(table[Products],λp.proj(var[p],price))
  # ⇝ <(10,<{:id=> 4, :name=>p1, :price=>10},     (Q_21)
  #          {:id=>13, :name=>p4, :price=>10}>),
  #     (20,<{:id=> 7, :name=>p2, :price=>20}>),
  #     (30,<{:id=> 9, :name=>p3, :price=>30}>)>
```

In step ① we derive each element in one stroke with its corresponding discriminator, which leads to the tabular representation in query q_{grp}. Based on the columns accommodating the discriminator, we derive new surrogate values that are eventually used to aggregate tied elements in the binding list e_{list}. In step ④ we prepare a single discriminator value (first tuple component) by eliminating the duplicates that have been introduced in step ①. The corresponding elements are then computed in ⑤; note how the column surr is used to establish new list identifiers. We conclude the rule with step ⑥ by assembling the column structure and surrogate list based on the information we calculated in step ②.

$$\frac{\begin{array}{l}\textcircled{1}\ \Gamma; \mathit{loop} \vdash \texttt{map(}e_{\text{list}}\texttt{,}\lambda x.\texttt{(}e_{\text{discr}}, x\texttt{))} \Rrightarrow (q_{\text{grp}}, (cs_{\text{discr}}, cs_x), \mathit{surr}_{\text{grp}}) \\ \textcircled{2} \begin{cases} l = |cs_{\text{discr}}| \quad cs'_x = cs_x \lhd l \quad \mathit{surr}'_{\text{grp}} = \mathit{surr}_{\text{grp}} \lhd l \\ \{\mathsf{item}_{\mathsf{x}_1}, \ldots, \mathsf{item}_{\mathsf{x}_\mathsf{m}}\} = \text{items}(cs_x) \end{cases} \\ \textcircled{3}\ q \equiv \vec{\Cup}_{\mathsf{surr}:\langle\mathsf{iter},\text{items}(cs_{\text{discr}})\rangle}(q_{\text{grp}}) \\ \textcircled{4}\ q_{\text{keys}} \equiv \pi_{\mathsf{iter},\mathsf{surr}:\{\mathsf{pos},\mathsf{item}_{\mathsf{l}+1}\},\text{items}(cs_{\text{discr}})}(\delta(\pi_{\mathsf{iter},\mathsf{surr},\text{items}(cs_{\text{discr}})}(q))) \\ \textcircled{5}\ q_{\text{groups}} \equiv \pi_{\mathsf{surr}:\mathsf{iter},\mathsf{pos},\mathsf{item}_{\mathsf{x}_1}:\mathsf{item}_1,\ldots,\mathsf{item}_{\mathsf{x}_\mathsf{m}}:\mathsf{item}_\mathsf{m}}(q) \\ \textcircled{6} \begin{cases} cs \equiv (cs_{\text{discr}}, \mathsf{item}_{\mathsf{l}+1}) \\ \mathit{surr} \equiv \{\mathsf{item}_{\mathsf{l}+1} \mapsto (q_{\text{groups}}, cs_x, \mathit{surr}'_{\text{grp}})\} \end{cases} \end{array}}{\Gamma; \mathit{loop} \vdash \texttt{group\_with(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{discr}}\texttt{)} \Rrightarrow (q_{\text{keys}}, cs, \mathit{surr})} \text{(LL-GroupWith)}$$

In **`partition`** $:: [\tau] \times (\tau \to \mathit{Bool}) \to [[\tau]]$ the elements in the input list are divided into exactly two lists based on the user supplied predicate. This function evaluates to a pair, in which elements for which the predicate evaluates to `false` end up in the list at the first position whereas the other elements are placed into the list at the second position.

$$\frac{\begin{array}{l} l = |cs_{\text{part}}| \\ \Gamma; \mathit{loop} \vdash \texttt{map(}e_{\text{list}}\texttt{,}\lambda x.\texttt{(}x, e_{\text{pred}}\texttt{))} \Rrightarrow (q_{\text{grp}}, (cs_{\text{part}}, \mathsf{item}_{\mathsf{l}+1}), \mathit{surr}_{\text{grp}}) \\ q \equiv \vec{\Cup}_{\mathsf{surr}:\langle\mathsf{iter},\mathsf{item}_{\mathsf{l}+1}\rangle}(q_{\text{grp}}) \\ q_{\text{keys}} \equiv \pi_{\mathsf{iter},\mathsf{surr}:\{\mathsf{pos},\mathsf{item}_1\}}(\delta(\pi_{\mathsf{iter},\mathsf{surr}}(q))) \\ q_{\text{groups}} \equiv \pi_{\mathsf{surr}:\mathsf{iter},\mathsf{pos},\text{items}(cs_x)}(q) \\ \mathit{surr} \equiv \{\mathsf{item}_1 \mapsto (q_{\text{groups}}, cs_x, \mathit{surr}_{\text{grp}})\} \end{array}}{\Gamma; \mathit{loop} \vdash \texttt{partition(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{pred}}\texttt{)} \Rrightarrow (q_{\text{keys}}, \mathsf{item}_1, \mathit{surr})} \text{(LL-Partition)}$$

3.15.6 Zip and Unzip

The function `zip` $:: ([\tau], [\sigma]) \to [(\tau, \sigma)]$ expects a pair containing two lists as argument and produces an output lists, in which every two elements from the input lists at the same position form a pair. In the following query the first list is shorter and the excess elements (b_4 and b_5) are discarded:

$$\begin{array}{c}\texttt{zip((<}a_1, a_2, a_3\texttt{>,<}b_1, b_2, b_3, b_4, b_5\texttt{>))} \\ \texttt{\#} \ \rightsquigarrow \ \texttt{<(}a_1, b_1\texttt{),(}a_2, b_2\texttt{),(}a_3, b_3\texttt{)>}\end{array} \quad (Q_{22})$$

Rule (LL-Zip) summarizes the following steps in a single inference rule: In step ①, following the translation of the argument, we use the proper components in the plan bundles that describe the argument lists (surr item_1 and surr item_2) to reconstruct the loop-lifted schema in step ②. It is critical to calculate the absolute positions for the elements of the argument lists in order to merge the queries q_1 and q_2 by means of an equi-join on the list identifier ($\mathsf{iter} = \mathsf{iter}_2$) and the position ($\mathsf{pos} = \mathsf{pos}_2$). In step ④, possibly exceeding elements from the longer list are removed from the surrogate map.

$$\dfrac{\begin{array}{l} ① \left\{\begin{array}{l} \Gamma; \mathit{loop} \vdash \texttt{(}e_1\texttt{,}e_2\texttt{)} \Rrightarrow (q, \texttt{(}\mathsf{item}_1, \mathsf{item}_2\texttt{)}, \mathit{surr}) \\ \mathit{surr}\ \mathsf{item}_1 = (q_1, cs_1, \mathit{surr}_1) \\ \mathit{surr}\ \mathsf{item}_2 = (q_2, cs_2, \mathit{surr}_2) \end{array}\right. \\ ② \left\{\begin{array}{l} l = |cs_1| \\ \{\mathsf{item}_{2\cdot 1}, ..., \mathsf{item}_{2\cdot \mathsf{n}}\} = \mathrm{items}(cs_2) \\ cs'_2 = cs_2 \rhd l \qquad \mathit{surr}'_2 = \mathit{surr}_2 \rhd l \end{array}\right. \\ ③ \left\{\begin{array}{l} q'_1 = \pi_{\mathsf{iter},\mathsf{pos2}:\mathsf{pos},\mathrm{items}(cs_1)}(\varrho_{\mathsf{pos}_2:\langle\mathsf{iter},\mathsf{pos}\rangle}(q_1)) \\ q'_2 = \pi_{\mathsf{iter}:\mathsf{iter2},\mathsf{pos2},\mathsf{item}_{2\cdot 1}:\mathsf{item}_{1+\mathsf{l}},\ldots,\mathsf{item}_{2\cdot\mathsf{n}}:\mathsf{item}_{\mathsf{n}+\mathsf{l}}}(\varrho_{\mathsf{pos}_2:\langle\mathsf{iter},\mathsf{pos}\rangle}(q_2)) \\ q_{\mathrm{zip}} = q'_1 \overset{=}{\bowtie}_{\mathsf{iter}=\mathsf{iter}_2 \wedge \mathsf{pos}=\mathsf{pos}_2} q'_2 \end{array}\right. \\ ④\ \mathit{surr}'_1 = \llbracket \mathit{surr}_1 \rrbracket^{\curlywedge}_{q_{\mathrm{zip}}} \qquad \mathit{surr}''_2 = \llbracket \mathit{surr}'_2 \rrbracket^{\curlywedge}_{q_{\mathrm{zip}}} \end{array}}{\Gamma; \mathit{loop} \vdash \texttt{zip((}e_1\texttt{,}e_2\texttt{))} \Rrightarrow (q_{\mathrm{zip}}, (cs_1, cs'_2), \mathit{surr}'_1 \uplus \mathit{surr}''_2)} \quad \text{(LL-Zip)}$$

As the inverse function to **zip()**, the function **unzip** $:: [(\tau, \sigma)] \rightarrow ([\tau], [\sigma])$ expects a list of pairs in order to transform it into a pair comprising a list of *first components* and a list of *second components*.

```
unzip(<(a₁,b₁),(a₂,b₂),(a₃,b₃)>)
  # ⇝ (<a₁,a₂,a₃>,<b₁,b₂,b₃>)
```

(Q_{23})

In rule (LL-Unzip), to place the components of the pairs into separate lists we need to split the hosting table into two parts based on the tuple components:

After the translation of the input list (step ①), we reconstruct the loop lifting schema in step ② by setting up separate column structures (cs_1 and cs'_2) and surrogate maps (surr_1 and surr_2). Additionally, these auxiliary structures help us to maintain the columns that are required to accommodate the respective tuple components.

In step ③, based on query q (derived in step ①), we calculate fresh surrogate values (column surr) in order to place the tuple components into separate lists. The queries q_1 and q_2 embody the resulting lists that respectively host the first tuple components and the second tuple components. The query q_{unzip} expresses the loop-lifted representation of the resulting pair that is linked with the queries q_1 and q_2 by means of the surrogate values.

In step ④, we prepare a fresh surrogate map (surr') that reflects the nested structure of the overall result.

$$\dfrac{\begin{array}{l} ① \; \Gamma; \mathit{loop} \vdash e \Rrightarrow (q, (cs_1, cs_2), \mathit{surr}) \\ ② \begin{cases} l = |cs_1| \\ \{\mathsf{item}_{2\cdot1}, \ldots, \mathsf{item}_{2\cdot\mathsf{n}}\} = \mathrm{items}(cs_2) \\ \mathit{surr}_1 = \mathit{surr}\,|_{\mathrm{items}(cs_2)} \\ cs_2' = cs_2 \lhd l \\ \mathit{surr}_2 = \mathit{surr}\,|_{\mathrm{items}(cs_1)} \lhd l \end{cases} \\ ③ \begin{cases} q' = \varrho_{\mathsf{surr}:\langle\mathsf{iter},\mathsf{pos}\rangle}(q) \\ q_1 \equiv \pi_{\mathsf{surr}:\mathsf{iter},\mathsf{pos},\mathrm{items}(cs_1)}(q') \\ q_2 \equiv \pi_{\mathsf{surr}:\mathsf{iter},\mathsf{pos},\mathsf{item}_{2\cdot1}:\mathsf{item}_1,\ldots,\mathsf{item}_{2\cdot\mathsf{n}}:\mathsf{item}_\mathsf{n}}(q') \\ q_{\mathrm{unzip}} \equiv \pi_{\mathsf{iter},\mathsf{pos},\mathsf{surr}:\{\mathsf{item}_1,\mathsf{item}_2\}}(q') \end{cases} \\ ④ \; \mathit{surr}' = \{\mathsf{item}_1 \mapsto (q_1, cs_1, \mathit{surr}_1), \mathsf{item}_2 \mapsto (q_2, cs_2', \mathit{surr}_2)\} \end{array}}{\Gamma; \mathit{loop} \vdash \mathtt{unzip(}e\mathtt{)} \Rrightarrow (q_{\mathrm{unzip}}, (\mathsf{item}_1, \mathsf{item}_2), \mathit{surr}')} \quad \text{(LL-UNZIP)}$$

3.15.7 MinBy and MaxBy

The higher-order function **min_by** $:: [\tau] \times (\tau \to \mathrm{atom}(\sigma)) \to \tau$ takes a list and a user-supplied function as arguments. The value in the input list that yields the minimum for the function is returned. If more than one value evaluates to the minimum, one of these values may be chosen arbitrarily.

To exemplify the semantics of function **min_by()** consider the following query that selects the list with the minimum length from the nested input list:

$$\begin{array}{l} \mathtt{min\_by(<<10,20,30>,<40,50>,<>>,}\lambda x.\mathtt{length}^{\mathsf{L}}\mathtt{(var[}x\mathtt{]))} \\ \quad \# \rightsquigarrow \mathtt{<>} \end{array} \qquad (Q_{24})$$

In rule (LL-MINBY) we take advantage of the logical order that may be established among the elements in the input list (e_{list}). The function **sort_by()** helps us to order the elements in ascending order according to the user-supplied function. The first element in this list is apparently the smallest element that must be returned. Consequently, we expand **min_by()** into the following definition to present its loop-lifted version.

$$\dfrac{\Gamma; \mathit{loop} \vdash \mathtt{first}^{\mathsf{L}}\mathtt{(sort\_by(}e_{\mathrm{list}}\mathtt{,}\lambda x.e_{\mathrm{body}}\mathtt{))} \Rrightarrow \mathit{pb}}{\Gamma; \mathit{loop} \vdash \mathtt{min\_by(}e_{\mathrm{list}}\mathtt{,}\lambda x.e_{\mathrm{body}}\mathtt{)} \Rrightarrow \mathit{pb}} \quad \text{(LL-MINBY)}$$

The converse function **max_by** $:: [\tau] \times (\tau \to \mathrm{atom}(\sigma)) \to \tau$ takes a list and a user-supplied function as arguments. Here, the value in the input list that assumes the maximum when evaluated by the function is returned. If more than one value yields the maximum, the choice for a result among these values is arbitrary.

The following query illustrates the semantics of **max_by()**. The list with the maximum length is returned:

```
max_by(<<10,20,30>,<40,50>,<>>,λx.length^L(var[x]))   (Q_25)
  # ⤳ <10,20,30>
```

Rule (LL-MaxBy) captures the function **`max_by()`**. The inference rule faithfully employs the function **`sort_by()`** to establish the logical order of the elements in the input list. Because this leads to a list in which the elements are in ascending order, we have to apply function **$\texttt{last}^{\mathsf{L}}$`()`** in order to obtain the maximum.

$$\frac{\Gamma; \mathit{loop} \vdash \texttt{last}^{\mathsf{L}}\texttt{(sort\_by(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{body}}\texttt{))} \Rrightarrow pb}{\Gamma; \mathit{loop} \vdash \texttt{max\_by(}e_{\text{list}}\texttt{,}\lambda x.e_{\text{body}}\texttt{)} \Rrightarrow pb} \text{(LL-MaxBy)}$$

3.16 Related Work

In their works [SS86; SS90; SS91], Scholl and Schek invest considerable effort to crystallize the similarities of the relational model and object-orientation to combine these seemingly diverging approaches into a unified framework. The authors give up the first-normal-form restriction to faithfully model hierarchic objects by means of nested relations. Enriched with "reference semantics" [Mey98], this approach even allows the definition of complex objects with non-hierarchical associations, which may be established by a classical foreign-key notation in the database back-end.

To manipulate objects, they propose a nested relational algebra, variants of which have been used to realize non-first normal form databases in the late eighties. Here, nesting of query language expressions may occur naturally to properly reflect the nested nature of the underlying object model while preserving the advantages of the declarative, set-oriented paradigm. Regarding the query aspect, this approach is very similar to what we propose in this work. The distinguishable characteristic, however, is that we directly expand on the results of [Bus01] and use a flat algebra to mimic the inherently nested surface language to meet the capabilities of today's (flat) SQL-centric back-ends. Furthermore we consider the order of the underlying data-structures throughout the entire calculation.

The LinQ project [Tor06; Sym06; BMT07; KB+07] integrates queries into several languages of the .Net suite. LinQ is designed to work with various data sources (including RDBMS) and provides a unified object-oriented interface with comprehension-flavored syntax to compose complex queries. Much like Switch, a suitable provider compiles an embedded LinQ query—represented as a chain of *Standard Query Operator* (SQO) invocations—into a sequence of SQL statements. Commonly, this entails and post-processing phases (much like ActiveRecord) performed on the host language heap. In its current implementation, LinQ still suffers from the n+1 query problem when establishing relationships between tables. Furthermore, all *Standard Query Operators* (SQO, for short) relying on order (*e.g.*

positional access) are currently not supposed to be database executable. A loop-lifted approach to implement the group of order-sensitive SQOs and to prevent the flood of look-alike queries has been proposed in [SB+10; GRS10].

Another route to language-integrated queries has been followed by Wiedermann and Cook. In [WC07] the authors present a technique based on *abstract interpretation* [CC77] for extracting set-oriented queries from imperative, object-oriented programs. This work has been refined in [IJ+09] and eventually led to *Remote Batch Invocation* (RBI), a language-level mechanism to identify *batches*, *i.e.* fragments that may be executed in a set-oriented fashion (possibly by a RDBMS). As the aforementioned approaches, the SQL generation approach used in RBI was unsatisfying under some circumstances. In a thriving cooperation, we were able to substitute the in-house SQL generation of RBI with a loop-lifted approach enabling us to unleash the full potential of this system [But11].

In [Coo09] Cooper presents a sound rewrite system to turn an iteration-based language operating over bags directly into SQL expression. A similar approach is employed in LINKS [CL+07], which also offers language-integrated queries in a unified web-framework solution (much like RAILS). A loop-lifted variant of LINKS has been proposed in [Ulr11].

In a different setting, Keller and Simons present a *Flattening Transformation* to efficiently compile vector-based, *nested* data-parallel programs into equivalent flat programs [KS96]. The intuition is to avoid the explicit source language iterator construct “apply-to-each” ($\{e \mid x \leftarrow e_{\text{in}}\}$), which allows nested computation. A series of simple source-level transformations are applied to obtain an iterator-free target language that only consists of primitive parallel operations.

To make up for the lack of iterators in the target language, the authors introduce a functional, called *lifting*. Applied on arbitrary function $f :: \tau \to \sigma$, the lifted variant $f^{\uparrow} :: \{\tau\} \to \{\sigma\}$ is operationally and semantically equivalent to $f^{\uparrow}(xs) = \{f x \mid x \leftarrow xs\}$. A function may be arbitrarily lifted ($f^{\uparrow n}$), but only the simple lifted functions can be found in the target language. In the first set of translation rules, all iterators are replaced by their equivalent “vectorwise functions” in a bottom-up walk over the expression tree. The flattening transformation then translates possibly occurring multiple-lifted functions into simple lifted functions present in the target language.

The above work shows a promising, alternative view of loop lifting. Currently, loop lifting directly translates a source-level construct into an equivalent algebraic plans. In a slightly different variation, the flattening transformation could be introduced to mediate between the inherently nested nature of SWITCH and the (flat) algebraic operators. The primitive parallel operations constituting the target language may be easily transferred into the relational domain.

Chapter 4

SQL Code Generation

In this chapter, we will take the last step towards the goal of turning relational database systems into highly efficient and scalable execution engines for arbitrary Switch expressions. For this purpose, we supplement the compilation process by a code generator that targets any SQL:1999-compliant database system. Proposed by Chamberlin and Boyce in nineteen and seventy four [CB74], SQL has established its dominance in the relational database area. The prevalence of SQL-centric systems provides ideal preconditions for our code generator to address the majority of actual RDBMS implementations available on the market. Additionally, this allows us to benefit significantly from their built-in optimizers to speed up the generated queries.

In the preceding chapter, we described how loop lifting may be used to turn a Switch expression into one or more DAG-shaped algebraic plans that jointly implement its dynamic semantics. The code generator then walks each of those intermediate algebraic plans and turns them into separate strictly standard-compliant SQL:1999 statements. In a sense, this approach makes relational algebra and SQL swap their traditional roles in query processing. The result is a compiler that can translate an arbitrary Switch expression into a representation executable on *any* SQL:1999-ready RDBMS. In a different setting, a variant of this approach has been described in [May07] in order to turn XQuery expressions into SQL queries.

In this context, the choice of relational algebra as intermediate language has its particular strengths. On the one hand, the semantics of relational operators are well-defined and oblivious of an actual database back-end. Compiler back-ends for the MonetDB column store and the kdb+ column store are described in [BG+06] and [Kan08] respectively. On the other hand, the algebraic primitives model the query capabilities of modern RDBMS sufficiently so that the generation of efficient SQL code remains feasible. To further facilitate the latter, the relational algebra has been designed with the processing capabilities of SQL-centric database kernels in mind. An example for this would be that the column projection (π) does not eliminate duplicate rows.

The set-oriented semantics of SQL blends well with the bulk primitives that constitute a relational expression. Furthermore, SQL condenses the semantics of several algebraic operators into a single `SELECT·FROM·WHERE` block. This observation led to a code generator approach that identifies *tiles* in the plans, similar to compilers for programming languages. Inside these tiles, we apply template instantiation to collapse a group of adjacent operators in the plan DAG into a single query. The SQL queries that result from the translation of one or more tiles are gathered and assembled into a single SQL:1999-compliant `WITH` statement that implements the overall semantics of a single query plan.

In the following section we will describe in which aspects the relational algebra deviates from SQL. Additionally, we provide a simple translation scheme to demonstrate the semantics of the relational bulk primitives. The problems arising with this approach lead to a refinement of the translation scheme to exploit both (1) the ability to merge several operators into a single SQL query (2) and the DAG structure of the algebraic plans. We will conclude the chapter by contributing the translation rules that incorporate the above techniques to turn query plans into efficient SQL queries that do not stumble over input data of considerable size.

4.1 Target Language: SQL:1999

Even though SQL and the relational algebra are both dedicated to the retrieval and manipulation of data within RDBMS, they diverge in several aspects:

Data Model.

The operators of the classical relational algebra are defined over relations that may be thought of as a set of tuples. In a relation, a tuple occurs not more than once and each operator is additionally obliged to ensure that tuples are eliminated from the result it produces.

The data model on which SQL operates is governed by the notion of a table that, in contrast to relations, may contain several instances of the same tuple (multiset or bag). The variant of the relational algebra we use in this work has been designed to respect the tabular model that dominates SQL-centric systems. If required, duplicate elimination (δ) is explicitly introduced into the query plans. In this regard, the primitives bear resemblance with the execution-plan operators perceivable in most database implementations.

Design.

The design of SQL was heavily influenced by both the relational algebra and the *relational tuple calculus*. The latter was proposed by Edgar F. Codd as an alternative declarative query language for the relational model [Cod70]. Like other declarative languages, SQL allows the user to describe the desired data without prescribing the operations that are necessary to produce the result. This separation between computation logic and the physical operations leaves it to the actual database implementation to plan, optimize and perform the steps that faithfully retrieve the queried data. An SQL query roughly subdivides into two language elements, including *clauses* and *expressions*.

Clauses are (possibly optional) constituent elements of a query. A clause comprises one or more expressions that may emit either an atomic value or a table. The type of the resulting value of an expression is determined by the enclosing clause. An expression that is used in the select clause, for example, must return an atom whereas the expressions in the from clause are constrained to generate a table.

As opposed to SQL, an expression composed of relational primitives dictates the data flow through the corresponding query plan. Due to the mathematical characteristics of the algebraic primitives these query plans are susceptible to a variety of optimizations. In database systems, SQL queries are typically translated into a notion similar to relational algebra, which assumes the role of an execution plan. These execution plans are then simplified according to a set of rewrite rules to enhance the performance of the queries. Another aspect that needs to be considered is that the relational operators are closed with respect to the tabular model. Each operator takes one or more tables and returns a table, so that the operators may be arbitrarily composed. This *closure property* of the relational operators is not easily transferable to SQL and deserve attention in the translation process.

Due to the compositionality of Switch, in which all constructs nest orthogonally as long as typing rules are obeyed, the query plans deviate from the well known π-σ-$\bowtie$ pattern that is generated by SQL compilers. The loop-lifted encoding typically leads to query plans comprising hundreds of operators that jointly implement the intricate semantics of a Switch expression.

4.1.1 A Simple Translation Scheme

In Table 4.1 we provide a simple translation scheme to convert a query plan into code executable on any SQL:1999-compliant system. Applied to the plan root, the translation walks the plan DAG in a bottom-up fashion and maps each operator

$sql[\![\pi_{a_1:b_1,\dots,a_n:b_n}(q)]\!]$	=	SELECT a_1 AS b_1,…, a_n AS b_n FROM ($sql[\![q]\!]$) AS C
$sql[\![\sigma_a(q)]\!]$	=	SELECT * FROM ($sql[\![q]\!]$) AS C WHERE a = true
$sql[\![@_{a:v}(q)]\!]$	=	SELECT *, v AS a FROM ($sql[\![q]\!]$) AS C
$sql[\![\delta(q)]\!]$	=	SELECT DISTINCT * FROM ($sql[\![q]\!]$) AS C

$sql[\![\varrho_{a:\langle b_1,\dots,b_n\rangle\|p}(q)]\!]$	=	SELECT *, ROW_NUMBER() OVER(
		PARTITION BY p
		ORDER BY $b_1,\dots,b_n$) AS a
		FROM ($sql[\![q]\!]$) AS C

$sql[\![\overrightarrow{\triangledown}_{a:\langle b_1,\dots,b_n\rangle}(q)]\!]$	=	SELECT *, DENSE_RANK() OVER(
		ORDER BY $b_1,\dots,b_n$) AS a
		FROM ($sql[\![q]\!]$) AS C

$sql[\![\circledcirc_{a:\langle b_1,b_2\rangle}(q)]\!]$	=	SELECT *, $b_1 \circ b_2$ AS a FROM ($sql[\![q]\!]$) AS C

$sql[\![q_1 \bar{\bowtie}_{a=b}\, q_2]\!]$	=	SELECT *
		FROM ($sql[\![q_1]\!]$) AS C1 INNER JOIN
		($sql[\![q_2]\!]$) AS C2 ON a = b

$sql[\![q_1 \times q_2]\!]$	=	SELECT * FROM ($sql[\![q_1]\!]$) AS C1, ($sql[\![q_2]\!]$) AS C2
$sql[\![q_1 \uplus q_2]\!]$	=	$sql[\![q_1]\!]$ UNION ALL $sql[\![q_2]\!]$
$sql[\![q_1 \;/\; q_2]\!]$	=	$sql[\![q_1]\!]$ EXCEPT ALL $sql[\![q_2]\!]$

$sql[\![\mathrm{GRP}_{a:\circ(b)/g}(q)]\!]$	=	SELECT g, $\circ(b)$ AS a
		FROM ($sql[\![q]\!]$) AS C
		GROUP BY g

$sql[\![\boxed{a_1 \dots a_n}]\!]$	=	SELECT $a_1,\dots,a_n$
		FROM (VALUES (…)) AS C($a_1,\dots,a_n$)

$sql[\![\mathsf{T}_{(a_1,\dots,a_n)\langle p\rangle}]\!]$	=	TABLE T

$sql[\![\text{♀}_{a_1,a_2,\langle a_1,\dots,a_n\rangle}(q)]\!]$	=	SELECT $a_3,\dots,a_n$
		FROM ($sql[\![q]\!]$) AS C
		ORDER BY a_1,a_2;

Table 4.1: Bottom-up translation of query plans into SQL:1999. Each relational operator is translated into a single SQL query. The query that results from the translation of a query plan is traversed by uncorrelated sub-queries and may be hardly simplified by the database optimizer.

to a single and complete SQL query. By planning a separate query for every single operator in the plan, we make up for the closure property that is not satisfied by SQL. The resulting table may then be consumed by the query that is introduced for the immediately following operator in the query plan.

The resulting SQL query reflects the underlying tree structure of the plan DAG and is thus pervaded by uncorrelated sub-queries. Figure 4.1 illustrates how the plan resulting from the loop-lifted translation of Query Q_{13} on page 94 is turned into an SQL query. While workable, this query displays a number of severe problems:

(i) The considerable size of the queries emitted by the translation scheme—even for small query plans—immediately leaps to the eye. As a Switch expression continues to grow, the corresponding SQL query may easily overwhelm the SQL compiler: Because each construct translates into a fixed number of relational operators, the number of primitives in a query plan grows linearly with the number of Switch constructs. Additionally, every operator in the query plan triggers the construction of a complete SQL query that nests as a sub-query in the overall `SELECT` statement.

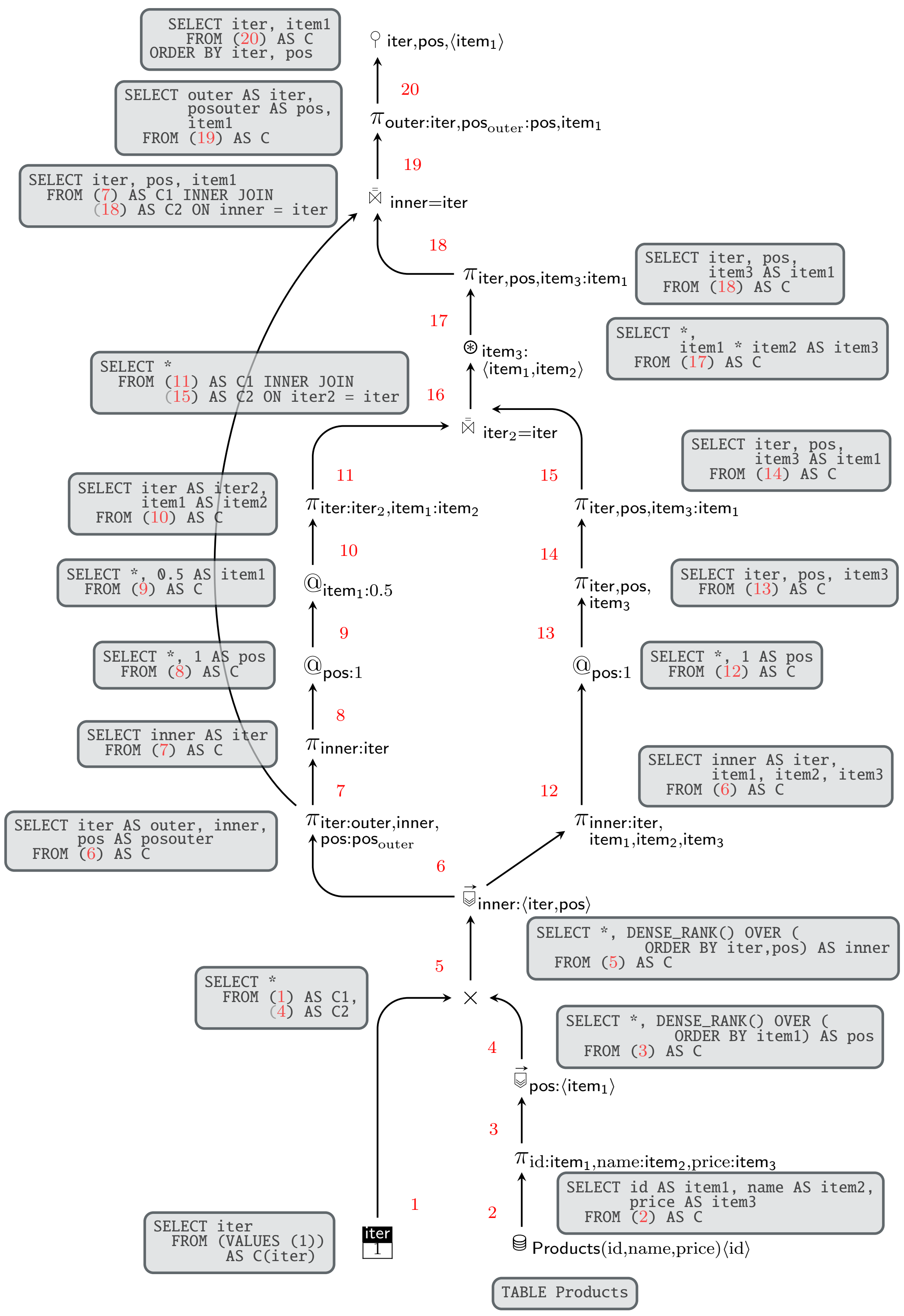
(ii) Groups of query operators that can be captured by a single `SELECT` statement are distributed over several uncorrelated sub-queries. Even though the removal of nested sub-queries is a problem that has been tackled in several works [BA+09; Kim82; Day87], these techniques have been adopted by few query optimizers.

(iii) The translation scheme does not consult any information about the DAG structure of the plans and leaves the identification of common sub-expression to the query optimizer.

This translation approach, however, already gives an impression of how SQL constructs may be used to implement the semantics of their algebraic counterparts. The resulting queries are reasonably "good natured" as, *e.g.* the fact that all `UNION` operations are over disjoint tables, nested queries in the `FROM` clause are uncorrelated and the occuring `JOIN` operations are equi-joins that implement the behavior of the nested iterations scopes.

4.2 Basic Techniques

The translation scheme we will devise in the following sections produces SQL queries that look more like handcrafted queries. We tackle the problems raised in the queries emitted by the translation scheme in Table 4.1, which overwhelmed any database optimizer we had on our workbench. For all back-ends the combi-



$\ominus$ iter,pos,⟨item$_1$⟩
20
$\pi_{\text{outer:iter,pos}_{\text{outer}}\text{:pos,item}_1}$
19
$\bowtie$ inner=iter
18
$\pi_{\text{iter,pos,item}_3\text{:item}_1}$
17
$\circledast$ item$_3$:⟨item$_1$,item$_2$⟩
16
$\bowtie$ iter$_2$=iter
11
15
$\pi_{\text{iter:iter}_2\text{,item}_1\text{:item}_2}$
$\pi_{\text{iter,pos,item}_3\text{:item}_1}$
10
14
@item$_1$:0.5
$\pi_{\text{iter,pos,item}_3}$
9
13
@pos:1
@pos:1
8
$\pi_{\text{inner:iter}}$
7
12
$\pi_{\text{iter:outer,inner,pos:pos}_{\text{outer}}}$
$\pi_{\text{inner:iter,item}_1\text{,item}_2\text{,item}_3}$
6
$\vec{\varrho}$ inner:⟨iter,pos⟩
5
$\times$
4
$\vec{\varrho}$ pos:⟨item$_1$⟩
3
$\pi_{\text{id:item}_1\text{,name:item}_2\text{,price:item}_3}$
1
2
iter
1
Products(id,name,price)⟨id⟩

```
  SELECT iter, item1
    FROM (20) AS C
ORDER BY iter, pos
```

```
SELECT outer AS iter,
       posouter AS pos,
       item1
  FROM (19) AS C
```

```
SELECT iter, pos, item1
  FROM (7) AS C1 INNER JOIN
       (18) AS C2 ON inner = iter
```

```
SELECT iter, pos,
       item3 AS item1
  FROM (18) AS C
```

```
SELECT *,
       item1 * item2 AS item3
  FROM (17) AS C
```

```
SELECT *
  FROM (11) AS C1 INNER JOIN
       (15) AS C2 ON iter2 = iter
```

```
SELECT iter, pos,
       item3 AS item1
  FROM (14) AS C
```

```
SELECT iter AS iter2,
       item1 AS item2
  FROM (10) AS C
```

```
SELECT *, 0.5 AS item1
  FROM (9) AS C
```

```
SELECT iter, pos, item3
  FROM (13) AS C
```

```
SELECT *, 1 AS pos
  FROM (8) AS C
```

```
SELECT *, 1 AS pos
  FROM (12) AS C
```

```
SELECT inner AS iter
  FROM (7) AS C
```

```
SELECT inner AS iter,
       item1, item2, item3
  FROM (6) AS C
```

```
SELECT iter AS outer, inner,
       pos AS posouter
  FROM (6) AS C
```

```
SELECT *, DENSE_RANK() OVER (
          ORDER BY iter,pos) AS inner
  FROM (5) AS C
```

```
SELECT *
  FROM (1) AS C1,
       (4) AS C2
```

```
SELECT *, DENSE_RANK() OVER (
          ORDER BY item1) AS pos
  FROM (3) AS C
```

```
SELECT id AS item1, name AS item2,
       price AS item3
  FROM (2) AS C
```

```
SELECT iter
  FROM (VALUES (1))
    AS C(iter)
```

```
TABLE Products
```

Figure 4.1: Illustrates the application of the simple translation scheme (Table 4.1) to Query Q_{13}. Observe that each relational primitive maps directly to a complete SQL query possibly occurs as sub-query in the formulation of the subsequent operator.

nation of query size and repeated sub-queries resulted in execution plans whose evaluation strategy largely reflect the structure of the underlying relational query plans derived in Chapter 3.

The SQL code generation approach we propose rests on two techniques that help the database optimizer to grasp and further simplify the SQL queries:

Tiling the Query Plans.
Here, we take the characteristics of the loop-lifted algebra and SQL into account to partition a query plan into tiles that may comprise several operators. The code generation then condenses all operators covered by a tile into a single query.

Identifying Repeated Subqueries.
Additionally, we consider the DAG shape of the plans to help the database optimizer to recognize repeated sub-queries in the resulting queries.

The combination of these techniques constitute the basic framework of the SQL code generation.

4.2.1 Tiling the Query Plans

In the translation approach we propose that the algebraic plans be chopped to let the RDBMS evaluate them in separate chunks. To identify suitable plan chunks, the code generator walks the plan DAG to find *tiles*, much like compilers for programming languages. In programming-language domain the technique that we adopted for the SQL generation is known as *Maximal Munch* [App04, ch. 9, p. 195].

In this particular context a tile is the maximum plan fragment that can be equally translated into a single SQL query. The goal is to cover the plan DAG with a minimum set of non-overlapping tiles. When the tiles are placed accordingly in the query plan, we apply template instantiation to collapse the operators for each tile into a single `SELECT` statement. In this regard the simple implementation in Table 4.1 expresses a special case of this approach in which each node is covered by a separate tile.

In our case, we try to find tiles that constitute as many operators as possible so that the corresponding SQL constructs still legally nest in a single query. The rigid skeleton of a `SELECT` query already prescribes how constructs need to be placed in the clauses to implement a specific relational operator. Essentially, each operator is captured by using a single clause or a combination of clauses that jointly implement its semantics on the relational back-end. The operator responsible for column renaming and projection (π), *e.g.*, only impacts the `SELECT` clause in the corresponding SQL statement whereas the SQL constructs for grouping (GRP that

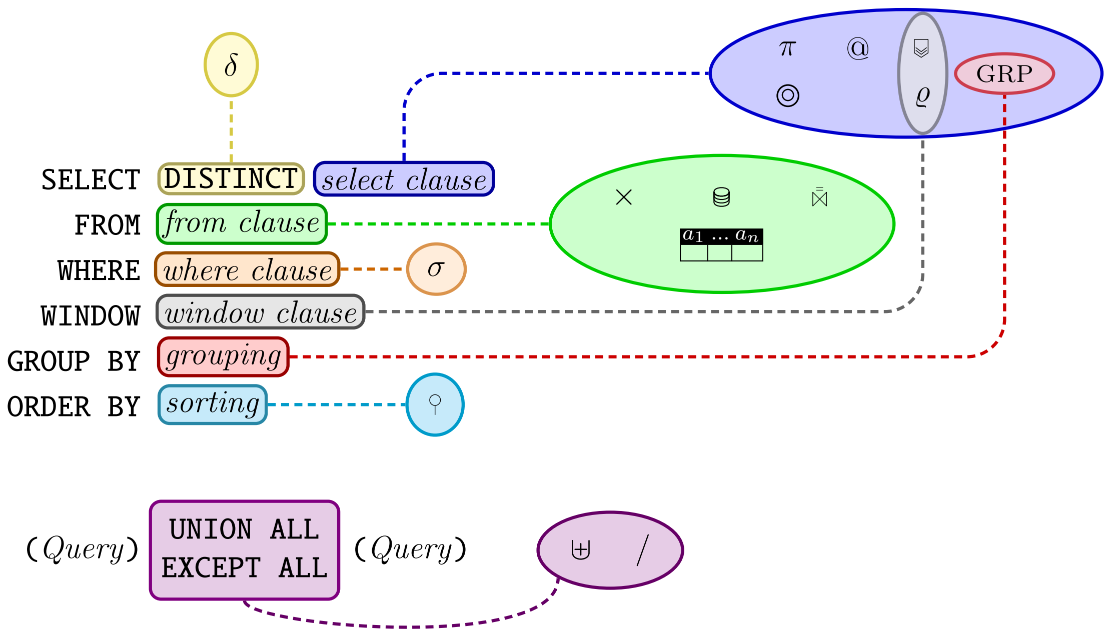


Figure 4.2: An SQL query captures the characteristics of each operator of the relational intermediate language by one or a combination of two clauses.

only co-occurs with an aggregation function) are situated at both the `GROUP BY` clause and the `SELECT` clause. Figure 4.2 demonstrates which clauses in SQL are used to implement the relational primitives. Note that the `WINDOW` clause, which we conveniently inlined in the translation scheme in Section 4.3, is listed explicitly.

On the one hand, it is important to see that the row-wise operators, which are hosted exclusively by the `SELECT` clause and `WHERE` clause as well as operators in the `FROM` clause do not necessarily complete a query when encountered in the query plan. On the other hand, the primitives that involve the window clause, grouping, sorting or duplicate elimination entail query termination since they generally need to consider a set of rows to faithfully complete their task. The `UNION` set operator and `EXCEPT` set operator require two tables that are identical in their structure to be combined (*union compatibility*). This observation leads to the classification of relational operators in Table 4.2, which reflects the placement in the SQL clauses.

In the translation rules we consult the above classification to set out the borders of the tiles. In the translation process we introduce an SQL query for each tile in the plan. In the initial query plan, each node is covered by a separate tile. To find the largest tile that fits we start at the leaf nodes of the query plan. If the current operator is found in the first group, we merge the tiles of the current and its adjacent operator. If, however, the operator is contained in the second

1. Merge Tiles			2. Terminate Tile				
π	$\bar{\bowtie}$	σ	δ	ϱ	GRP	♀	$\uplus$
@	$\times$			⛉			/
$\circledcirc$	$\begin{array}{\|c\|c\|c\|}\hline a_1 & \dots & a_n \\ \hline & & \\ \hline \end{array}$						
	⛁		$\curlyvee$ (Section 4.2.2)				

Table 4.2: Classification of the relational primitives according to their placement in the clauses of an SQL query. We will use this classification to determine where tiling takes place in the query plan.

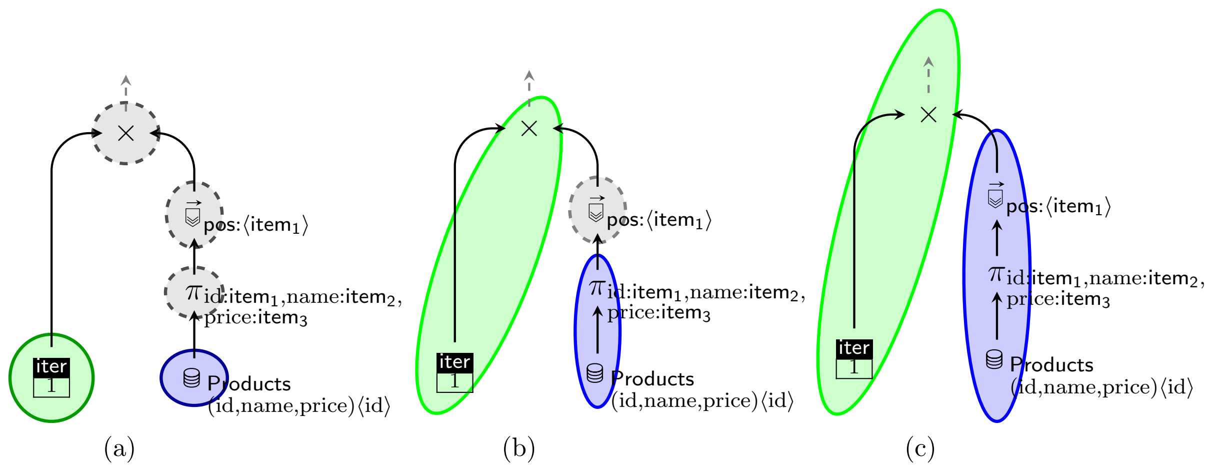


Figure 4.3: To find tiles, we start at the leaves of the query plan. The operators covered by gray tiles are not visited yet. The classification in Table 4.2 helps us to decide whether the tile of the current operator may be combined with the tile covering the adjacent operator in the query plan. The rank operator (⛉) terminates a tile so that the operators covered by the blue area in (c) result in a single SQL query. The cross-product ($\times$) does not terminate the green tile, which continues to enfold further operators along the line.

group, we terminate the tile. In both cases the process restarts at the following operator. Figure 4.3 exemplifies how this process operates on a fragment of the plan in Figure 4.1.

4.2.2 Identifying Repeated Sub-Queries

The loop-lifted compilation turns any SWITCH expression into one ore more DAG-shaped algebraic plans. Typically, these plans exhibit a wealth of sharing opportunities that may be exploited in the SQL translation. In the underlying DAG, such sharing opportunities appear in form of a branch. For the operator at which the plan starts to fork into two or more strands this means that the resulting intermediate relation is used more than once in the subsequent manipulations. When merged into a single `SELECT`-query, this may easily lead to repeated sub-queries that are hardly recognized by the query optimizer in order to be factored out and executed once instead of multiple times.

So far, the opportunity to share common sub-expressions has not been considered in the generated query plans. In this way, a naive translation scheme that walks a query plan in a bottom-up fashion to assemble an SQL query would inevitably lead to repeated sub-queries, as this is the case for the translation scheme in Table 4.1. Figure 4.4a depicts a plan fragment that turns up in the loop-lifted translation of conditionals. Starting at the column projection operator (π), the plan divides into two separate strands, so that the intermediate relation is used by both adjacent filter operators (σ).

To properly reflect this situation in the SQL-centric back-end, we introduce the "materialization" operator $\curlyvee_{a_1,\dots,a_n}(q)$, which, applied to query q, explicitly tells us to

(i) apply some sort of materialization strategy in the back-end in order to
(ii) prevent a costly re-computation of q by means of a repeated sub-query.

Note how the plan in Figure 4.4b is "rewired" to interpose materialization between the adjacent operators, so that the filter operators directly refer to the materialized result of the column projection operator.

Woven into the algebraic plan, materialization does not affect the underlying semantics of a query, and may be perceived as identity operator so that the equivalence

$$\curlyvee_{a_1,\dots,a_n}(q) = q$$

holds for all plan fragments q. For a seamless integration into an algebraic plan, the *live columns*

$$\mathit{cols}[\![q]\!] = \{a_1,\dots,a_n\} \ ,$$

which the plan fragment q emits, are adopted by the materialization primitive. The live columns may be obtained in a single plan walk over q, as illustrated in

Table 4.3. On the back-end, however, this operator is critical to ensure that the resulting SQL queries reflect the substantial degree of sharing in the DAG-shaped algebraic plans.

Operator		**Live Columns**
$cols\llbracket \pi_{a_1:b_1,\dots,a_n:b_n}(q) \rrbracket$	$=$	$\{b_1, \dots, b_n\}$
$cols\llbracket \sigma_a(q) \rrbracket$	$=$	$cols\llbracket q \rrbracket$
$cols\llbracket @_{a_1:v_1,\dots,a_n}(q) \rrbracket$	$=$	$cols\llbracket q \rrbracket \uplus \{a_1, \dots, a_n\}$
$cols\llbracket \delta(q) \rrbracket$	$=$	$cols\llbracket q \rrbracket$
$cols\llbracket \varrho_{a:\langle b_1,\dots,b_n\rangle}(p)q \rrbracket$	$=$	$cols\llbracket q \rrbracket \uplus \{a\}$
$cols\llbracket \vec{\Cup}_{a:\langle b_1,\dots,b_n\rangle}(q) \rrbracket$	$=$	$cols\llbracket q \rrbracket \uplus \{a\}$
$cols\llbracket \circledcirc_{a:\langle b_1,b_2\rangle}(q) \rrbracket$	$=$	$cols\llbracket q \rrbracket \uplus \{a\}$
$cols\llbracket q_1 \overset{=}{\bowtie}_{a=b} q_2 \rrbracket$	$=$	$cols\llbracket q_1 \rrbracket \uplus cols\llbracket q_2 \rrbracket$
$cols\llbracket q_1 \times q_2 \rrbracket$	$=$	$cols\llbracket q_1 \rrbracket \uplus cols\llbracket q_2 \rrbracket$
$cols\llbracket q_1 \uplus q_2 \rrbracket$	$=$	$cols\llbracket q_1 \rrbracket$
$cols\llbracket q_1 \,/\, q_2 \rrbracket$	$=$	$cols\llbracket q_1 \rrbracket$
$cols\llbracket \mathrm{GRP}_{a:\circ(b)/g}(q) \rrbracket$	$=$	$\{a, g\}$
$cols\llbracket \begin{array}{\|c\|c\|c\|}\hline a_1 & \dots & a_n \\ \hline & & \\ \hline\end{array} \rrbracket$	$=$	$\{a_1, \dots, a_n\}$
$cols\llbracket \text{⛁}_{\mathsf{T}(a_1,\dots,a_n)\langle p\rangle} \rrbracket$	$=$	$\{a_1, \dots, a_n\}$
$cols\llbracket \text{♀}_{a_1,a_2,\langle a_1,\dots,a_n\rangle}(q) \rrbracket$	$=$	$\{a_1, \dots, a_n\}$
$cols\llbracket \curlyvee_{a_1,\dots,a_n}(q) \rrbracket$	$=$	$\{a_1, \dots, a_n\}$

Table 4.3: Deriving the live columns of an operator in a bottom-up traversal.

To formalize the above intuition, we regard a query plan as a direct acyclic graph (Q, A). The set Q maintains the set of algebraic primitives that participate in the evaluation of the query plan. A is comprised of pairs $A \subseteq Q \times Q$ and denotes the set of arcs between operators. Each pair hosts the origin and the target of an arc in the first and the second tuple component respectively.

Given a query graph (Q, A), we define the *vicinity* for an operator $q \in Q$

$$\nu(q) \coloneqq \{r \mid (q, r) \in A\} \tag{4.1}$$

as the set of adjacent operators that are immediately reached when following the direction of an arc starting at q. Using Eq. (4.1), branches may be easily identified by their degree

$$\deg^+(q) \coloneqq |\nu(q)| \ . \tag{4.2}$$

For a table operator $q \in Q$ a degree

$$\deg^+(q) > 1$$

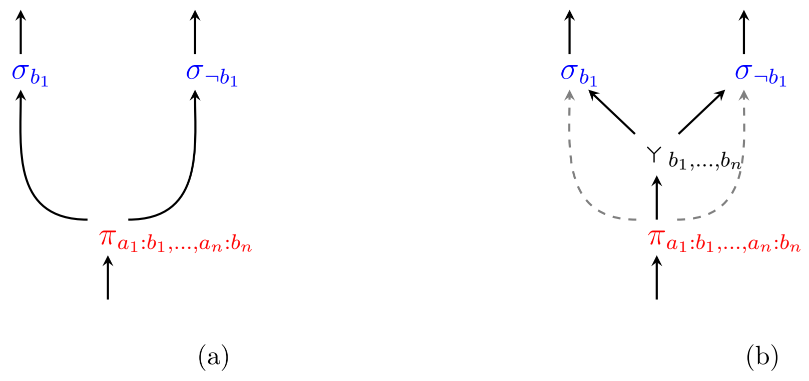

σ_{b_1}
$\sigma_{\neg b_1}$
$\pi_{a_1:b_1,\ldots,a_n:b_n}$
(a)
σ_{b_1}
$\sigma_{\neg b_1}$
$\curlyvee_{b_1,\ldots,b_n}$
$\pi_{a_1:b_1,\ldots,a_n:b_n}$
(b)

Figure 4.4: The red operator in (a) denotes a repeated subexpression that is referenced by both following operators. In (b) we interpose $\curlyvee_{a_1,\ldots,a_n}$ between the adjacent operators to prescribe some sort of materialization in the back-end.

identifies a branch in the underlying query plan.

Based on Eqs. (4.1) and (4.2), from a query plan (Q, A) we derive the enriched query plan $(Q_{\text{mat}}, A_{\text{mat}})$ that explicitly tells us where materialization is to take place in the SQL-centric back-end in two steps:

Identifying Branches.
Building upon the degree (Eq. (4.2)), we isolate operators and arcs where the plan splits into two or more separate strands:

$$A_{\curlyvee} := \{(q, r) \mid (q, r) \in A, \deg^{+}(q) > 1\} \tag{4.3}$$
$$Q_{\curlyvee} := \{q \mid (q, r) \in A_{\curlyvee}\} \ . \tag{4.4}$$

Rewiring the Plan.
In the second step, we append the materialization operators to the set of table operators

$$Q_{\text{mat}} := Q \uplus \{\curlyvee_{cols[\![q]\!]} \mid q \in Q_{\curlyvee}\} \ , \tag{4.5}$$

and properly rewire the plan components accordingly to interpose the materialization operator between the adjacent operators.

$$\begin{aligned} A_{\text{mat}} \ &:= \ (A - A_{\curlyvee}) \\ &\uplus \ \{(q, \curlyvee_{cols[\![q]\!]}) \mid q \in Q_{\curlyvee}\} \\ &\uplus \ \{(\curlyvee_{cols[\![q]\!]}, r) \mid (q, r) \in A_{\curlyvee}\} \ . \end{aligned} \tag{4.6}$$

Having derived the enriched plan ($Q_{\text{mat}}, A_{\text{mat}}$), we may now leverage this information in the back-end. Since the materialization primitive does not prescribe a specific materialization strategy, we are free to tailor the strategy to a specific database system at a later juncture. In this work, however, we decided to employ the SQL:1999 `WITH` clause for this purpose, as we will describe in the following section. A strategy that uses *temporary tables* [MS01, ch. 4, p. 100] or *views*[MS01, ch. 4, p106] has been described in [GM+07].

4.2.3 Common Table Expressions

With the techniques described above, we introduce several tiles into a query plan. Each tile embodies a separate SQL query that needs to be executed on the back-end. The resulting table is possibly used by other queries to correctly implement the semantics of the underlying query plan. So far, we extensively used sub-queries to assemble the queries into a single SQL query.

Introduced in SQL:1999, common table expressions (or `WITH` clauses) enables us to factor out common sub-expressions. One or more "inner" queries may be bound to a *query name* that, listed in the `WITH` clause, can be referenced in the definition of the "outer" query expression (see Figure 4.5b). The query engine executes such queries exactly once in order to arrange the result into a *virtual table*. This virtual table is then actually used in each place where it is referenced [MS01, ch. 9, p. 296ff].

```
SELECT ...
  FROM (Q) AS T1
     , (Q) AS T2
WHERE ...
```

(a) When not recognized by the optimizer, Query Q is executed twice.

```
WITH
  t AS (Q)
SELECT ...
  FROM (t) AS T1
     , (t) AS T2
 WHERE ...
```

(b) Query Q is bound to a query name (t) that is referenced in the "outer" query expression.

Figure 4.5: On the right-hand side, the `WITH`-clause executes the repeated sub-expression (Q) only once instead multiple times as in the case depicted in (a).

The `WITH` clause is readily available in most database systems [Witc; Wita; Witb]. In contrast to temporary tables, the `WITH` clause has no persistent side effect on the database system. In the SQL code generation we gather the SQL queries for each tile to immediately bind them to a query name that is then used to reference the resulting table. Because we employ common table expressions for both tiling and factoring out repeated sub-queries, we may safely remove the materialize operator from operator constellations of the form $\mathsf{Y}(\circ(q))$ (with $\circ \in \{\delta, \varrho, ⛉, \text{GRP}, \uplus, /\}$) from the loop-lifted query plans in order to prevent a superfluous binding.

4.3 Translation Rules

In this section we provide the translation rules that integrate the techniques discussed in the above section into a single translation scheme. The SQL dialect that is generated in the course of this process is illustrated in Table 4.4. The primary elements of the translation scheme are rules of the form

$$q \Mapsto \langle (\Sigma, \mathcal{F}, w), \mathcal{C} \rangle \quad ,$$

which translate an expression from relational plan fragment (q) into the following components:

1. The first tuple component embodies a `SELECT·FROM·WHERE` block in SQL. Each component maintains the decisive components to instantiate a query that can be extended by an optional clause to correctly implement the semantics of an operator.

   (i) The column map (Σ) maps column names used in the loop-lifted algebra to SQL expressions of the type *coldef* (see Table 4.4). The expressions may be used in both the select clause and the where clause. Note that *column references* are instantly bound to a *correlation name*, which helps us to maintain distinct column names throughout the translation.

   (ii) The set $\mathcal{F}$ contains the query expressions that may be used in the `FROM` clause of a SQL expression. As in the above case, correlation names are bound to these expressions to prevent name clashes. In the following, we refer to this component as *source expressions.*

   (iii) The boolean expression w is used to store a conjunctive predicate list. We use w to phrase the `WHERE` clause when a concrete SQL query is instantiated. In the following, we refer to this tuple component as *conjunction list.*

   All three components provided, we can easily employ template instantiation

$$\text{sfw}(\delta, \Sigma, \mathcal{F}, w) \equiv \begin{array}{rl} \texttt{SELECT} & \overbrace{\texttt{DISTINCT}}^{\text{if } \delta \text{ is set}} \ \{\Sigma\, a \ \texttt{AS} \ a \mid a \in \text{dom}(\Sigma)\} \\ \texttt{FROM} & \mathcal{F} \\ \texttt{WHERE} & w \end{array}$$

to assemble an SQL query. To express the `SELECT` clause, we bind the expressions to the column names found in the domain of the partial function ($\mathrm{dom}(\Sigma)$). Duplicate elimination (δ) is only enforced if the optional first parameter is set.

2. The second component ($\mathcal{C}$) embodies the set of query definitions that are possibly used in the calculation of the overall expression in terms of a `WITH` clause. Whenever a basic block is identified in the SQL translation, the corresponding SQL query is gathered and bound to a query name that may be used as a reference in the remaining translation.

In the remaining section we consider each operator in the loop-lifted algebra separately to provide the proper translation rule that turns each of them into a SQL query. Applied to the plan root, these rules assemble a `WITH` statement comprising several queries that jointly implement the semantics of the query plan. In Figure 4.6 we demonstrate how the translation rules, we provide in the following sections, affect the translation of Query Q_{13}.

4.3.1 Value Expressions

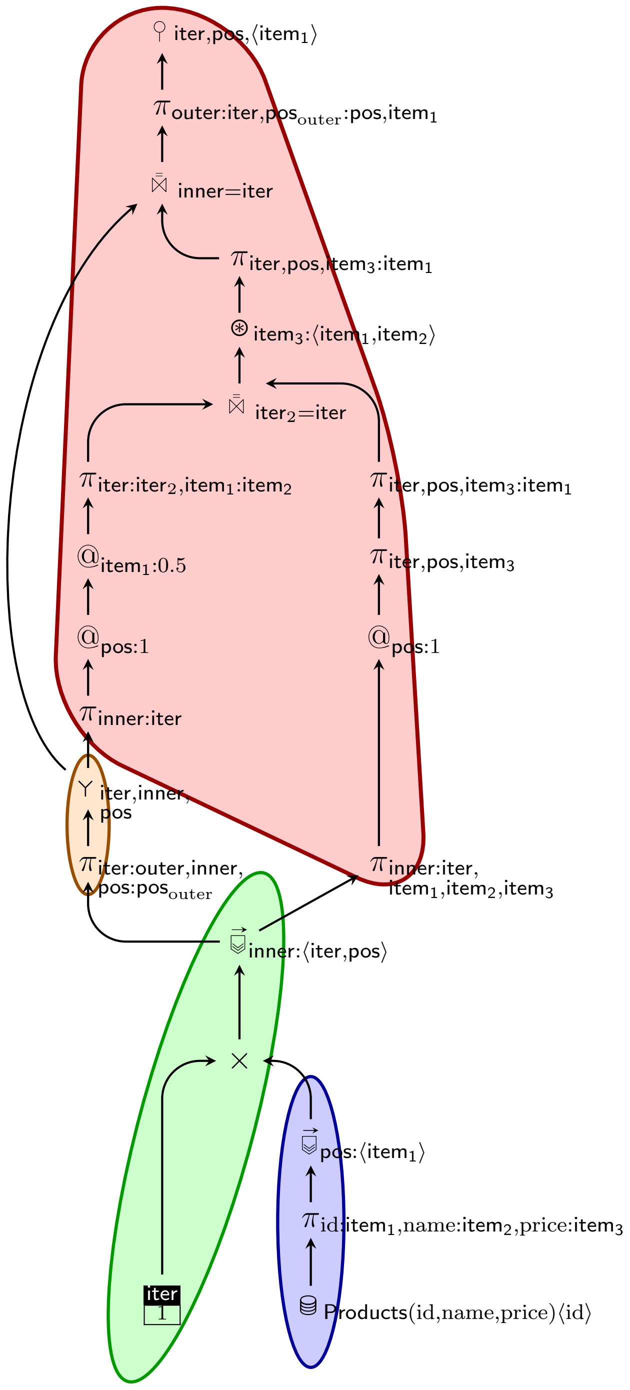
Each of the following relational primitives (π,σ,$\circledcirc$) is translated into a *value expression* that may be used in any clause of an SQL statement to properly reflect the semantics of an operator. We gather these expressions and maintain them in the column map (Σ). The expressions remain accessible for further use by referencing the column name that is provided by the relational operators.

In rule (SQL-Project), to capture column projection and renaming, we set up the column map to maintain the columns that are of continuing interest in the translation. In this way, the operator turns into a compile time operation. The abundance of projection operators, which were introduced to prevent name clashes in the loop-lifted query plans, are rendered superfluous. SQL permits us to avoid ambiguity by establishing qualified column references using *correlation names*. If a correlation name is specified for a table, any reference to a column must use the correlation name rather than the table name.

$$\frac{\begin{array}{c} q \Mapsto \langle(\Sigma, \mathcal{F}, w), \mathcal{C}\rangle \\ \Sigma' \equiv \{b_1 \mapsto \Sigma\, a_1, \ldots, b_n \mapsto \Sigma\, a_n\} \end{array}}{\pi_{a_1:b_1,\ldots,a_n:b_n}(q) \Mapsto \langle(\Sigma', \mathcal{F}, w), \mathcal{C}\rangle} \quad \text{(SQL-Project)}$$

In a similar fashion we translate the algebraic primitives for column attachment (@), unary and binary operators ($\circledcirc$). In rule (SQL-Attach), rule (SQL-Unary) and rule (SQL-Binary) this is respectively reflected by extending the column map by

```
WITH
C2(item1,item2,item3,pos) AS (
  SELECT C1.id AS item1,
         C1.name AS item2,
         C1.price AS item3,
         DENSE_RANK() OVER (
           ORDER BY id) AS pos
    FROM Products AS C1),
C4(inner,iter,pos,item1,item2,item3) AS (
  SELECT DENSE_RANK() OVER (
           ORDER BY C3.iter, C2.pos) AS inner,
         C3.iter AS iter, C2.pos AS pos,
         C2.item1 AS item1, C2.item2 AS item2,
         C2.item3 AS item3
    FROM (VALUES (1)) AS C3(iter), C2),
C5(outer,inner,posouter) AS (
  SELECT C4.iter AS outer, C4.inner,
         C4.pos AS posouter
    FROM C4)
  SELECT C6.outer AS iter,
         C7.item3 * 0.5 AS item1
    FROM C5 AS C6, C4 AS C7, C5 AS C8
   WHERE C6.inner = C7.inner
     AND C7.inner = C8.inner
ORDER BY C6.outer, C6.posouter
```



iter,pos,⟨item$_1$⟩
π outer:iter,pos$_{\text{outer}}$:pos,item$_1$
inner=iter
π iter,pos,item$_3$:item$_1$
item$_3$:⟨item$_1$,item$_2$⟩
iter$_2$=iter
π iter:iter$_2$,item$_1$:item$_2$
π iter,pos,item$_3$:item$_1$
@ item$_1$:0.5
π iter,pos,item$_3$
@ pos:1
@ pos:1
π inner:iter
iter,inner,pos
π iter:outer,inner,pos:pos$_{\text{outer}}$
π inner:iter,item$_1$,item$_2$,item$_3$
inner:⟨iter,pos⟩
×
pos:⟨item$_1$⟩
π id:item$_1$,name:item$_2$,price:item$_3$
iter
1
Products(id,name,price)⟨id⟩

Figure 4.6: Translation of Query Q_{13} that relies on the rules we provide in the following section. We leverage tiling to condense several algebraic primitives into a single SQL query. Additionally, the identification of sub-queries helps us to prevent costly recalculations.

Expressions

withexpr	:=	`WITH` $\overline{qdef}$ *qexpr*
qdef	:=	*tabname*`(`$\overline{col}$`)` `AS` `(`*qexpr*`)`
qexpr	:=	*qexpr* `UNION ALL` *qexpr*
	|	*qexpr* `EXCEPT ALL` *qexpr*
	|	*sfw*
sfw	:=	`SELECT DISTINCT` $\overline{coldef}$ `FROM` $\overline{tabexpr}$ `WHERE` *pred* [*group*] [*order*]
coldef	:=	*expr* `AS` *col*
pred	:=	*pred* `AND` *pred*
	|	*expr* *comp* *expr*
	|	`true` | `false`
comp	:=	`=` | `<` | `>` | `<=` | . . .
expr	:=	*corr*`.`*col* | *val*
	|	*expr* *arithop* *expr*
	|	*pred*
	|	*ranking*
	|	*setfunc*`(`*expr*`)`
setfunc	:=	`MAX` | `MIN` | . . . | `EVERY`
ranking	:=	`ROW_NUMBER() OVER (` [*part*] [*order*] `)`
	|	`DENSE_RANK() OVER (` [*order*] `)`
part	:=	`PARTITION BY` *expr*
order	:=	`ORDER BY` $\overline{expr}$
group	:=	`GROUP BY` $\overline{expr}$
arithop	:=	`+` | `-` | `*` | `/` | . . .
tabexpr	:=	*tabname* `AS` *corr*
	|	`(VALUES (`$\overline{val}$`)) AS` *corr*`(`$\overline{col}$`)`
val	:=	`1` | . . . | *n*
	|	`'...'`

Table 4.4: SQL:1999 dialect used in the translation of query plans. The brackets mark optional query fragments. Overlined expressions $\overline{e}$ stand for $e_1, \ldots, e_n$.

the respective *value expressions*. For example, the binary operators (⊕, ⊖, . . .) are directly mapped to their readily available SQL counterparts (+, -, . . .).

$$\frac{q \mapsto \langle(\Sigma, \mathcal{F}, w), \mathcal{C}\rangle}{@_{a:v}(q) \mapsto \langle(\Sigma \uplus \{a \mapsto v\}, \mathcal{F}, w), \mathcal{C}\rangle} \quad \text{(SQL-Attach)}$$

$$\frac{\circ \in \{\texttt{+}, \texttt{-}, \texttt{!}\} \qquad q \mapsto \langle(\Sigma, \mathcal{F}, w), \mathcal{C}\rangle}{\circledcirc_{a:\langle b\rangle}(q) \mapsto \langle(\Sigma \uplus \{a \mapsto \circ \Sigma\, b\}, \mathcal{F}, w), \mathcal{C}\rangle} \quad \text{(SQL-Unary)}$$

$$\frac{\begin{array}{c}\circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{|}, \texttt{\&}, \texttt{==}, \texttt{<>}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}\} \\ q \mapsto \langle(\Sigma, \mathcal{F}, w), \mathcal{C}\rangle\end{array}}{\circledcirc_{a:\langle b_1, b_2\rangle}(q) \mapsto \langle(\Sigma \uplus \{a \mapsto \Sigma\, b_1 \circ \Sigma\, b_2\}, \mathcal{F}, w), \mathcal{C}\rangle} \quad \text{(SQL-Binary)}$$

4.3.2 Predicates

The select operator (σ_a) confines a table to rows that satisfy a given condition. In rule (SQL-Select), to capture this behavior, we obtain the predicate from the column map ($\Sigma\, a$), which, in turn is employed to refine the boolean expression (w) by means of a conjunction (**AND**). When materialized, the boolean expression is used to phrase the **WHERE** clause of an SQL statement.

$$\frac{q \mapsto \langle(\Sigma, \mathcal{F}, w), \mathcal{C}\rangle}{\sigma_a(q) \mapsto \langle(\Sigma, \mathcal{F}, w \ \texttt{AND} \ \Sigma\, a), \mathcal{C}\rangle} \quad \text{(SQL-Select)}$$

4.3.3 Working with (Multiple) Tables

The operators in this section are converted into table expressions that may be integrated into the **FROM** clause of an SQL query. When encountered in the query plan, they are turned into a *table expression* that is collected in the source expression, which represents the from clause.

In the translation rule (SQL-Cross), which captures the cross product (×) the source expressions ($\mathcal{F}_1$ and $\mathcal{F}_2$) of the translated constituent plan fragments are merged by means of a disjoint union. Note that even if a table expression occurs twice in the resulting set, they are still distinguishable by their correlation name.

$$\frac{q_i \mapsto \langle(\Sigma_i, \mathcal{F}_i, w_i), \mathcal{C}_i\rangle\,\big|_{i=1,2}}{q_1 \times q_2 \mapsto \langle(\Sigma_1 \uplus \Sigma_2, \mathcal{F}_1 \uplus \mathcal{F}_2, w_1 \wedge w_2), \mathcal{C}_1 \wedge \mathcal{C}_2\rangle} \quad \text{(SQL-Cross)}$$

Rule (SQL-EqJoin), which captures the equi-join operator ($\bowtie_{a=b}$) is turned into a *classic comma-separated join*. The resulting cross product additionally confines the expression to satisfy the join condition $\Sigma\, a = \Sigma\, b$. It is important to see that the cardinality of the source expressions ($\mathcal{F}_1$ and $\mathcal{F}_2$) may possibly comprise more than one table expression. For this reason we cannot use the `INNER JOIN` construct.

$$\frac{q_1 \times q_2 \Mapsto \langle(\Sigma, \mathcal{F}, w), \mathcal{C}\rangle}{q_1 \overset{=}{\bowtie}_{a=b} q_2 \Mapsto \langle(\Sigma, \mathcal{F}, w \ \texttt{AND} \ \Sigma\, a = \Sigma\, b), \mathcal{C}\rangle} \ (\text{SQL-EqJoin})$$

To capture a table reference (⛁) that occurs in the query plan, we initialize the source expression with the proper table name. Note that, to prevent name clashes between columns, the table name as well as the columns are instantly bound to a fresh correlation name. Rule (SQL-TableRef) reflects this situation. The conjunction list is initialized accordingly to let every row in this table pass.

$$\frac{\begin{array}{c} c = \text{is a fresh correlation name} \\ \Sigma \equiv \{a_1 \mapsto c\texttt{.}a_1, \ldots, a_n \mapsto c\texttt{.}a_n\} \end{array}}{⛁_{\mathsf{T}(a_1,\ldots,a_n)\langle p\rangle} \Mapsto \langle(\Sigma, \{\texttt{T AS } c\}, \texttt{true}), \mathcal{C}\rangle} \ (\text{SQL-TableRef})$$

A similar strategy is applicable to the remaining operators in this section. In rule (SQL-Table), to establish a table literal in the SQL-centric back-end, we use the *table value constructor* (`VALUES`). The contents of the table literal are straightforwardly adopted. An empty table may be constructed by obeying to rule (SQL-TableEmpty). The `NULL` values are used to ensure adherence to the SQL standard, even though the will never be exposed due to the conjunctive list that evaluates to `false`.

$$\frac{\begin{array}{c} t = \text{is a fresh table name} \\ \mathcal{F} \equiv \left\{\begin{array}{l} \texttt{(VALUES (}v_{1\cdot 1}\texttt{,}\ldots\texttt{,}v_{1\cdot n}\texttt{),}\ \ldots \\ \qquad\qquad\quad \texttt{(}v_{m\cdot 1}\texttt{,}\ldots\texttt{,}v_{m\cdot n}\texttt{)) AS } t\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{)} \end{array}\right\} \end{array}}{\begin{array}{|c|c|c|} \hline a_1 & \ldots & a_n \\ \hline v_{1\cdot 1} & \ldots & v_{1\cdot n} \\ \vdots & \vdots & \vdots \\ v_{m\cdot 1} & \ldots & v_{m\cdot n} \\ \hline \end{array} \Mapsto \langle(\{a_1 \mapsto t\texttt{.}a_1, \ldots, a_n \mapsto t\texttt{.}a_n\}, \mathcal{F}, \texttt{true}), \mathcal{C}\rangle} \ (\text{SQL-Table})$$

$$\frac{\begin{array}{c} t = \text{is a fresh table name} \\ \mathcal{F} \equiv \left\{\texttt{(VALUES (null,}\ldots\texttt{,null) AS } t\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{))}\right\} \end{array}}{\begin{array}{|c|} \hline a_1 \ldots a_n \\ \hline \emptyset \\ \hline \end{array} \Mapsto \langle(\{a_1 \mapsto t\texttt{.}a_1, \ldots, a_n \mapsto t\texttt{.}a_n\}, \mathcal{F}, \texttt{false}), \mathcal{C}\rangle} \ (\text{SQL-TableEmpty})$$

4.3.4 Duplicate Elimination

To eliminate duplicates in a table, SQL provides the keyword `DISTINCT`, which may be specified in the `SELECT` clause of a query. In rule (SQL-Distinct) we instantiate an SQL query and apply duplicate elimination to all visible columns. The query is then bound to a query name to provide a reference for subsequent operations.

$$\frac{\begin{array}{c} q \Mapsto \langle(\Sigma, \mathcal{F}, w), \mathcal{C}\rangle \\ t = \text{is a fresh table name} \qquad c = \text{is a fresh correlation name} \\ \mathcal{C}' \equiv \{t\texttt{(}\mathrm{dom}(\Sigma)\texttt{) AS (}\mathrm{sfw}(\delta, \Sigma, \mathcal{F}, w)\texttt{)}\} \end{array}}{\delta(q) \Mapsto \langle(\{a \mapsto c\texttt{.}a \mid a \in \Sigma\}, \{t \texttt{ AS } c\}, \texttt{true}), \mathcal{C} \uplus \mathcal{C}'\rangle} \quad (\text{SQL-Distinct})$$

To establish duplicate elimination in the back-end, the cost-based optimizer may choose between a hash-based or a sort-based execution strategy. PostgresSQL, *e.g.*, prefers hashing if the hash table is likely to fit into the working memory and order is not relevant for the final result. A sort-based approach, however, is favored whenever the preservation of order is likely to lead to a cheaper access-plan.

4.3.5 Ranking

The `ROW_NUMBER()` and `DENSE_RANK()` OLAP ranking facilities are employed to manipulate tabular representations of lists. Because we use row ranks for list positions, surrogate values and grouping, such rank-based operations are encountered frequently in the query plans. Most perceivable implementations of these operators introduce a blocking sort operation into the execution plans, which presents a serious cost for actual RDBMS implementations.

res	i_1	i_2
1	1	1
2	1	3
3	1	4
1	4	1
2	4	1

(a) `ROW_NUMBER() OVER ( PARTITION BY` i_1 `ORDER BY` i_2`)`

res	i_1	i_2
1	1	1
2	1	3
3	1	4
4	4	1
5	4	1

(b) `ROW_NUMBER() OVER ( ORDER BY` i_1,i_2`)`

res	i_1	i_2
1	1	1
2	1	3
3	1	4
4	4	1
4	4	1

(c) `DENSE_RANK() OVER ( ORDER BY` i_1,i_2`)`

Figure 4.7: Semantics of SQL/OLAP ranking facilities `ROW_NUMBER()` and `DENSE_RANK()`. The column res respectively contains the result of the below expressions.

In rule (SQL-Rownum) and rule (SQL-Rank), we use the *inline window clause* [Mel02, ch. 7, p. 310] rather than the explicit alternative that is specified in a separate clause. The `WINDOW` clause permits us to define suitable partitions and ordering among the rows of the resulting table. For a demonstration of the semantics of `ROW_NUMBER()` and `DENSE_RANK()` consider Figure 4.7.

$$\frac{\begin{array}{c} q \Mapsto \langle (\Sigma, \mathcal{F}, w), \mathcal{C} \rangle \\ \Sigma' \equiv \Sigma \uplus \left\{ a \mapsto \begin{array}{l} \texttt{ROW\_NUMBER() OVER (PARTITION BY } \Sigma\, p \\ \qquad\qquad\qquad\quad \texttt{ORDER BY } \Sigma\, a_1 \texttt{,} \ldots \texttt{,} \Sigma\, a_n \texttt{)} \end{array} \right\} \\ t = \text{is a fresh table name} \qquad c = \text{is a fresh correlation name} \\ \mathcal{C}' \equiv \{ t\texttt{(}\mathrm{dom}(\Sigma')\texttt{) AS (}\mathrm{sfw}(\Sigma', \mathcal{F}, w)\texttt{)} \} \end{array}}{\varrho_{b:\langle a_1,\ldots,a_n \rangle \| p}(q) \Mapsto \langle (\{ a \mapsto c\texttt{.}a \mid a \in \Sigma' \}, \{ t \texttt{ AS } c \}, w), \mathcal{C} \uplus \mathcal{C}' \rangle} \quad \text{(SQL-Rownum)}$$

$$\frac{\begin{array}{c} q \Mapsto \langle (\Sigma, \mathcal{F}, w), \mathcal{C} \rangle \\ \Sigma' \equiv \Sigma \uplus \{ a \mapsto \texttt{DENSE\_RANK() OVER (ORDER BY } \Sigma\, a_1 \texttt{,} \ldots \texttt{,} \Sigma\, a_n \texttt{)} \} \\ t = \text{is a fresh table name} \qquad c = \text{is a fresh correlation name} \\ \mathcal{C}' \equiv \{ t\texttt{(}\mathrm{dom}(\Sigma')\texttt{) AS (}\mathrm{sfw}(\Sigma', \mathcal{F}, w)\texttt{)} \} \end{array}}{\vec{\Cup}_{b:\langle a_1,\ldots,a_n \rangle}(q) \Mapsto \langle (\{ a \mapsto c\texttt{.}a \mid a \in \Sigma' \}, \{ t \texttt{ AS } c \}, w), \mathcal{C} \uplus \mathcal{C}' \rangle} \quad \text{(SQL-Rank)}$$

4.3.6 Grouping

In the loop-lifted algebra the group operator (GRP) additionally entails the calculation of an aggregate function on the input groups. The aggregate functions in the algebra (MAX(),MIN(),...) are mapped to their equivalent *set functions* in SQL (`MAX()`, `MIN()`,...) [MS01, ch. 5, p. 132].

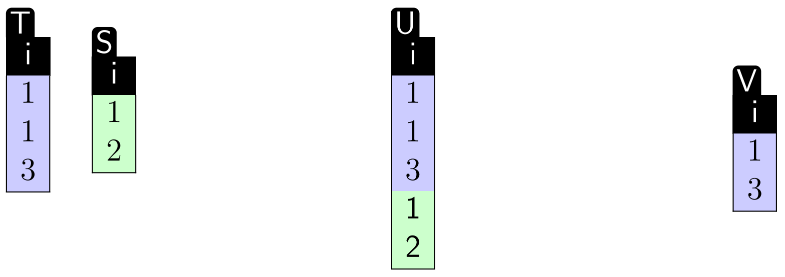
Similar to duplicate elimination, the cost-based optimizer of the RDBMS needs to weigh up whether to introduce a hash-based approach or a sort-based approach in the execution plan to divide the input table into groups. The set functions in turn consider each group separately to complete their task.

$$\frac{\begin{array}{c} \circ \in \{\mathrm{MAX}, \mathrm{MIN}, \mathrm{SUM}, \mathrm{AVG}, \mathrm{COUNT}, \mathrm{EVERY}\} \quad q \Mapsto \langle (\Sigma, \mathcal{F}, w), \mathcal{C} \rangle \\ t = \text{is a fresh table name} \qquad c = \text{is a fresh correlation name} \\ \mathcal{C}' \equiv \{ t\texttt{(}b\texttt{,}g\texttt{) AS (}\mathrm{sfw}(\Sigma \uplus \{ b \mapsto \circ(\Sigma\, a) \}, \mathcal{F}, w) \texttt{ GROUP BY } \Sigma\, g\texttt{)} \} \end{array}}{\mathrm{GRP}_{b:\circ(a)/g}(q) \Mapsto \langle (\{ b \mapsto c\texttt{.}b, g \mapsto c\texttt{.}g \}, \{ t \texttt{ AS } c \}, \texttt{true}), \mathcal{C} \uplus \mathcal{C}' \rangle} \quad \text{(SQL-Group)}$$

4.3.7 Set Operators

The algebraic set operators ($\uplus$ and $/$) are straightforwardly mapped to their counterparts in the SQL centric back-end (`UNION` and `EXCEPT`). Both set operations require their input table to be *union compatible.* Hence, the tables must have the same number of columns, and the data types of each pair of columns in the two tables that appear at the same relative positions must be comparable. In rule (SQL-Union) and rule (SQL-Except), to ensure that this restriction is met, we instantiate the SQL queries for both constituent operators.

Because we perform set operations only over disjoint tables, we may faithfully employ the low-cost variants `UNION ALL` and `EXCEPT` on the back-end. The set operator `UNION` operates similarly to `UNION ALL`, but entails duplicate elimination



(a) input tables T and S (b) U = T `UNION ALL` S (c) V = T `EXCEPT` S

Figure 4.8: Semantics of set operators in SQL. Note that `UNION ALL` does not entail duplicate elimination in the result table. By using `EXCEPT` we avoid removing duplicates from the input tables before set difference takes place.

on the resulting table. Similarly, `EXCEPT ALL` removes all duplicates from the input tables before the rows are transfered into the result table. The semantics of the set operations is exemplified in Figure 4.8.

$$\frac{\begin{array}{cc}\text{dom}(\Sigma_1) = \text{dom}(\Sigma_2) & q_i \Mapsto \langle(\Sigma_i, \mathcal{F}_i, w_i), \mathcal{C}_i\rangle\big|_{i=1,2} \\ t = \text{is a fresh table name} & c = \text{is a fresh correlation name} \\ \multicolumn{2}{c}{\mathcal{C}' \equiv \{t\texttt{(}\text{dom}(\Sigma_1)\texttt{) AS (}\text{sfw}(\Sigma_1, \mathcal{F}_1, w_1)\ \texttt{UNION ALL}\ \text{sfw}(\Sigma_2, \mathcal{F}_2, w_2)\texttt{)}\}}\end{array}}{q_1 \uplus q_2 \Mapsto \langle(\{a \mapsto c\texttt{.}a \mid a \in \Sigma_1\}, \{t\ \texttt{AS}\ c\}, \texttt{true}), \mathcal{C}_1 \uplus \mathcal{C}_2 \uplus \mathcal{C}'\rangle} \quad \text{(SQL-UNION)}$$

$$\frac{\begin{array}{cc}\text{dom}(\Sigma_1) = \text{dom}(\Sigma_2) & q_i \Mapsto \langle(\Sigma_i, \mathcal{F}_i, w_i), \mathcal{C}_i\rangle\big|_{i=1,2} \\ t = \text{is a fresh table name} & c = \text{is a fresh correlation name} \\ \multicolumn{2}{c}{\mathcal{C}' \equiv \{t\texttt{(}\text{dom}(\Sigma_1)\texttt{) AS (}\text{sfw}(\Sigma_1, \mathcal{F}_1, w_1)\ \texttt{EXCEPT}\ \text{sfw}(\Sigma_2, \mathcal{F}_2, w_2)\texttt{)}\}}\end{array}}{q_1 \mathbin{/} q_2 \Mapsto \langle(\{a \mapsto c\texttt{.}a \mid a \in \Sigma_1\}, \{t\ \texttt{AS}\ c\}, \texttt{true}), \mathcal{C}_1 \uplus \mathcal{C}_2 \uplus \mathcal{C}'\rangle} \quad \text{(SQL-EXCEPT)}$$

4.3.8 Serialization

The serialization operator (♀) marks the root of a query plan and triggers the arrangement of a complete SQL query out of the fragments that have been collected during the plan walk. Therefore, the formulation in rule (SQL-SERIALIZE) is twofold:

(i) The set of query expression ($\mathcal{C}$)—each representing a single tile in the query plan—is turned in to a comma-separated list of the following shape:

$$\{qe\texttt{,} \mid qe \in \mathcal{C}\} = \begin{array}{c} qn_1\ \texttt{AS (}sql_1\texttt{),} \\ \vdots \\ qn_m\ \texttt{AS (}sql_m\texttt{),} \end{array}$$

In this form the query name qn_i maybe used as a reference for an SQL query sql_i (with $i \in \{1, \ldots, n\}$). The back-end's execution engine ensures that each SQL query in the list is executed not more than once, even if there exist multiple references to a single query name.

(ii) The column map (Σ), the source expressions ($\mathcal{F}$), and the conjunctive list (w) are converted into an SQL query that forms the overall result. To facilitate the serialization in the host language, the result is additionally ordered according to the list identifiers (column a_1) and the positions (column a_2).

$$\frac{q \Mapsto \langle(\Sigma, \mathcal{F}, w), \mathcal{C}\rangle}{\begin{array}{ll} & \texttt{WITH} \\ ♀_{a_1,a_2,\langle a_3,\ldots,a_n\rangle}(q) \Mapsto & \{qe\texttt{,} \mid qe \in \mathcal{C}\} \\ & \mathrm{sfw}(\Sigma, \mathcal{F}, w) \\ & \texttt{ORDER BY } \Sigma\, a_1\texttt{, } \Sigma\, a_2 \end{array}} \quad \text{(SQL-Serialize)}$$

We use the ordered table, calculated by the above query, and the type of the respective Switch expression to construct the proper values in the host language. If the Switch expression is supposed to evaluate to a list, the iteration values assume the role of list identifiers while the position values determine the position of the elements in the list. If the Switch expression leads to a nested result, the loop-lifted translation produces two or more query plans leading to two or more SQL queries. To properly place nested values, we rely on the surrogate values that we introduced in the loop-lifted translation scheme for this purpose. In the host language we use a kind of *sort merge join* to restore the result.

4.3.9 Explicit Binding

Whereas the loop-lifted translation features column projection and renaming (π) to prevent name clashes, in the SQL generation approach we use the possibility to introduce correlation names into the queries to ensure distinct column names within an SQL query. In this way, the column projection does not affect the query runtime.

An important observation is that if a query plan divides into two or more strands these strands (originating in $\curlyvee$), may possibly coalesce into a single operator. In the example in Figure 4.9, the query plan illustrates ta self-join in which the correlation name, used to reference the result of materialization, occurs twice. To guarantee distinct correlation names, we introduce fresh correlation names whenever the materialization operator is visited for the second time in the plan walk.

The above observation leads to an approach where the operator assumes a different kind of semantics during the translation. When the materialize operator

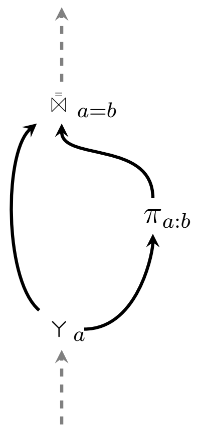


Figure 4.9: When not handled properly, the above constellation of plan operators leads to a self-join that let the SQL compiler stumble over ambiguous column names due to identical correlation names.

is visited for the first time in the plan walk, we instantiate the SQL query for its constituent operator. To properly reflect the DAG structure of the query plan, this query takes place among the query expressions ($\mathcal{C}$). Rule (SQL-MAT-1) expresses this behavior. The materialize operator is then substituted by

$$\curlyvee^{\langle(\Sigma,\mathcal{F},w),\mathcal{C}\rangle} \quad .$$

The results from the translation of the materialize operator ($\langle(\Sigma,\mathcal{F},w),\mathcal{C}\rangle$) are annotated to handle further visits. Additionally, we generate a new correlation name to prevent name clashes for possibly coalescing plan strands. Rule (SQL-MAT-2) formalizes this situation. Consider the red fragments in rule (SQL-MAT-1) that are adopted in the formulation of rule (SQL-MAT-2).

$$\frac{\begin{array}{c} q \Mapsto \langle(\Sigma,\mathcal{F},w),\mathcal{C}\rangle \\ t = \text{is a fresh table name} \qquad c = \text{is a fresh correlation name} \\ \mathcal{C}' \equiv \{t\texttt{(}\mathrm{dom}(\Sigma)\texttt{)}\ \texttt{AS}\ \texttt{(}\mathrm{sfw}(\Sigma,\mathcal{F},w)\texttt{)}\} \\ \Sigma^\bullet \equiv \{a \mapsto c\texttt{.}a \mid a \in \Sigma\} \qquad \mathcal{F}^\bullet \equiv \{t\ \texttt{AS}\ c\} \qquad w^\bullet \equiv \texttt{true} \\ \mathcal{C}^\bullet \equiv \mathcal{C} \uplus \mathcal{C}' \end{array}}{\curlyvee_{a_1,\ldots,a_n}(q) \Mapsto \langle(\Sigma^\bullet,\mathcal{F}^\bullet,w^\bullet),\mathcal{C}^\bullet\rangle} \quad \text{(SQL-MAT-1)}$$

$$\frac{\begin{array}{c} c' = \text{is a fresh correlation name} \\ \Sigma \equiv \{a_1 \mapsto c\texttt{.}a_1,\ldots,a_n \mapsto c\texttt{.}a_n\} \qquad \mathcal{F} \equiv \{t\ \texttt{AS}\ c\} \qquad w \equiv \texttt{true} \\ \Sigma' \equiv \{a_1 \mapsto c'\texttt{.}a_1,\ldots,a_n \mapsto c'\texttt{.}a_n\} \qquad \mathcal{F}' \equiv \{t\ \texttt{AS}\ c'\} \end{array}}{\curlyvee^{\langle(\Sigma^\bullet,\mathcal{F}^\bullet,w^\bullet),\mathcal{C}^\bullet\rangle} \Mapsto \langle(\Sigma',\mathcal{F}',w),\mathcal{C}\rangle} \quad \text{(SQL-MAT-2)}$$

$$\frac{\begin{array}{c} q \doteq q' \mapsto \langle(\Sigma, \mathcal{F}, w), \mathcal{C}\rangle \\ t = \text{is a fresh table name} \qquad c = \text{is a fresh correlation name} \\ \mathcal{C}' \equiv \{t\texttt{(}\mathrm{dom}(\Sigma)\texttt{)}\ \texttt{AS}\ \texttt{(}\mathrm{sfw}(\Sigma, \mathcal{F}, w)\texttt{)}\} \\ \Sigma' \equiv \{a \mapsto c\texttt{.}a \mid a \in \Sigma\} \qquad \mathcal{F}' \equiv \{t\ \texttt{AS}\ c\} \qquad w' \equiv \texttt{true} \end{array}}{\curlyvee_{a_1,\ldots,a_n}(q) \doteq \curlyvee^{\langle(\Sigma',\mathcal{F}',w'),\mathcal{C}\rangle} \mapsto \langle(S', F', W'), \mathcal{C} \uplus \mathcal{C}'\rangle} \ \text{(SQL-Mat-Subst)}$$

$$\frac{\begin{array}{c} q \doteq q' \mapsto \langle(\Sigma, \mathcal{F}, w), \mathcal{C}\rangle \\ \Sigma' \equiv \{b_1 \mapsto \Sigma\, a_1, \ldots, b_n \mapsto \Sigma\, a_n\} \end{array}}{\pi_{a_1:b_1,\ldots,a_n:b_n}(q) \doteq \pi_{a_1:b_1,\ldots,a_n:b_n}(q') \mapsto \langle(\Sigma', \mathcal{F}, w), \mathcal{C}\rangle} \ \text{(SQL-Project-Subst)}$$

Table 4.5: By the above rules we exemplify how the translation rules can be refined to reflect plan rewrites. Beside delivering the proper SQL translation for q the judgment $q \doteq q' \mapsto \langle(\Sigma, \mathcal{F}, w), \mathcal{C}\rangle$ additionally tells us to substitute the plan operator q by q' in the query plan.

Since plan rewrites are confined to the materialize operator, we do not reflect rewrite semantics in the translation rules to improve readability. However, in Table 4.5 we provide two examples to illustrate how the translation rules can be refined to integrate plan rewrites.

Chapter 5

Assessment

In this chapter we will closely look at the benefits we obtain when using SWITCH to formulate queries. We will see that deep query embedding and SQL code generation can still lead to competitive query performance. In fact, SWITCH-generated SQL code can often contend with the manually written SQL, as we will demonstrate for a diversity of query classes and use cases taken from *Spree*. To make this point, we dissect the queries focusing on the comparison of SWITCH's SQL output with

(1) the SQL statement sequence generated by ACTIVERECORD 3.0.5 and with

(2) a handcrafted SQL query variant that possibly entails calculation on the RUBY heap in order to restore host-language data structures.

In all experiments, we measure the complete round-trip time, which includes (i) the compilation time required to produce proper SQL statements as well as (ii) query shipping, (iii) result shipping, and (iv) materialization on the host language heap.

Moreover, the inspection of the execution plans will help us to understand and appreciate the efforts that relational database systems (DB2® in this case) take to efficiently implement the tested queries. The techniques applied by DB2 range from accelerating data access via indexes to comprehensive rewrites that let the optimizer reinvent the execution strategy based on a thorough cardinality estimation of participating plan operators. The optimizer especially benefits from data distribution statistics available in the catalog tables.

For the quantitative assessment, we leverage the similarity between *Spree*'s data model and the TPC-H benchmark [Tpc] to derive application data for various scale factors. The measured round-trip time will give us an impression of how DB2 can cope with the workloads. All measurements include the time required to bring the expressions into a database executable format and to materialize the result on the RUBY heap.

Before we jump into the experimentation part of this chapter, we will briefly discuss the optimization potential of the loop-lifted query plans. Here, we will benefit from the assembly-style semantics of the relational primitives to perform plan rewrites that possibly lead to drastic simplifications. The optimization approach presented in [Rit10] and [GMR09] is discussed with regard to XQUERY, but is as well applicable to the plan shapes generated by SWITCH.

5.1 Optimization in a Nutshell

Even with the code generation approach we detailed in the previous chapter, current commercial SQL optimizers typically struggle with the generated queries. The unusual query shapes stemming from the loop-lifted compilation scheme feature the scattered distribution of equi-joins, duplicate elimination, numbering operators and grouping. These peculiarities call for an optimizer that is designed to exploit the intricacies of the nested, ordered data model.

The optimization approach we rest on in this work pursues the goal to separate a *join graph* from the blocking operators in the *plan tail*. The bundles of base table references and joins constituting a join graph along with the blocking operators in the plan let the relational optimizer face a problem known inside-out. Based on its data statistics, the cost-based optimizer may then autonomously work out the best execution plan to deliver the result.

To establish the desired plan shapes, the optimizer adopts a local rewrite technique coined *peephole optimization* [McK65]. This technique considers only a small set of algebraic operators to perform optimizations. Several attribute grammars pass operator properties across the query plan in a single top-down or bottom-up walk to support this local rewrite strategy. Each optimization rule considers one or more properties in order to simplify the plan. All optimization rules may be divided into the following three heuristics [Rit10, ch. 4, p. 47]:

House Cleaning. Superfluous plan fragments are pruned and useless operators that are guaranteed to yield an empty table are removed from a query plan. Relational primitives are simplified based on key and constant information. Variants of selection and projection pushdowns are performed and common sub-plans removed.

Order Minimization. Certain operator constellations lead to the removal of order constraints from the query plans, implemented by row numbering and ranking operators. For example, a ranking operator that is exclusively used by the serialize operator to ensure the order of the overall result may be faithfully removed from the query plan.

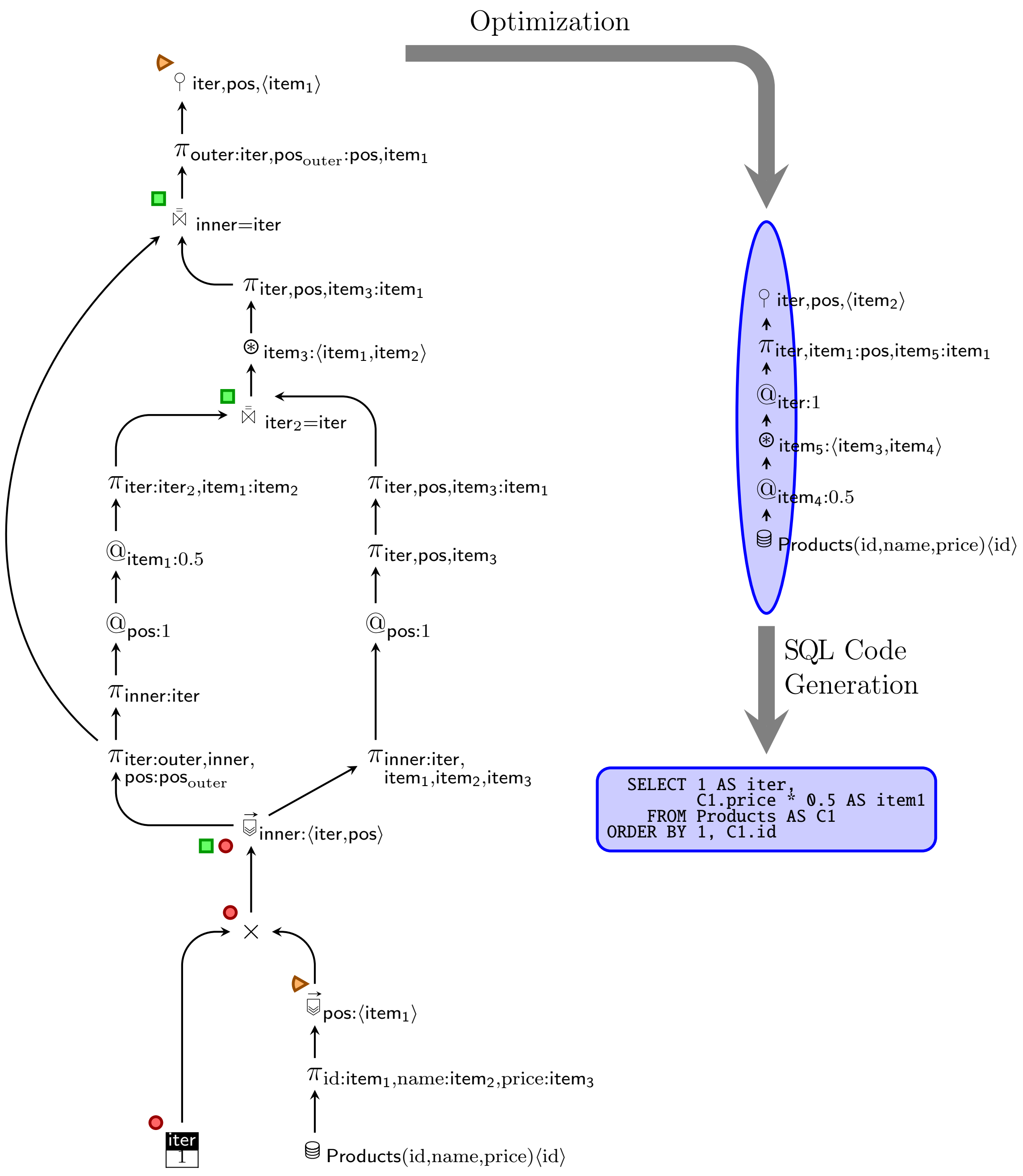
Query Unnesting. Several equi-joins that are introduced in the loop-lifted translation are rendered useless by the optimizer. Particularly the *mapping joins*, which reflect the iterative semantics of a Switch expression exhibit various properties that make them susceptible to further optimization.

In the rest of this section we will briefly discuss how optimization affects the loop-lifted query plan for Query Q_{13}. In Figure 5.1 we compare the original query plan that directly results from the loop-lifted translation (on the left hand side) with the plan that is rewritten according to the heuristics we sketched above (on the right hand side). While operationally equivalent to the original plan, the optimized plan uses neither numbering operators nor equi-joins to successfully implement the iterative, ordered semantics of the underlying Switch expression. To achieve this, the optimizer performs the following principal simplifications (respective operators that are affected in the query plan are marked by ●, 🟩 and ▷):

● To infer constant values, the optimizer walks the query plan in a bottom-up fashion [Rit10, Ch. 4, p. 52]. At the cross product this property is considered to recognize that the iter column originating in the loop table exposes a single constant value. This tree pattern may thus be replaced straightforwardly by a column attachment operator ($@_{\mathsf{iter}:1}$).
The rank operator (⊌) that is directly following the cross product also benefits from the constant property inference. Due to the constancy the column iter does not affect this operator and may thus be removed.

🟩 In most cases, the loop lifting compilation introduces equi-joins to combine table columns that were split earlier in the compilation process to implement iteration. To remove the aligning equi-joins, the optimizer performs a series of *equi-join pushdowns* to bring joins towards this *split point* [Rit10, Ch. 4, p. 78ff], where it may be removed if possible.
In this case, the rank operator embodies this split point. At this point, the optimizer becomes aware that both join columns originate in the same column (inner). Additionally, the inner column is marked as key column [Rit10, Ch. 4, p. 54], which leads to the elimination of the equi-join operator. Consequently, the rank operator is now rendered superfluous because column inner remains unreferenced by the upstream plan operators.

▷ The last remaining rank operator in the query plan is also subject to removal. The column pos that is produced as a result is exclusively used in the serialize operator (♀) to ensure the order of the final result. Apparently, column item_1, which is the only column that influences the logical order of the final result, may be directly used in the serialization operator [Rit10, Ch. 4, p. 73ff].



```
  SELECT 1 AS iter,
         C1.price * 0.5 AS item1
    FROM Products AS C1
ORDER BY 1, C1.id
```

Figure 5.1: Comparison of the original query plan of Query Q_{13} (on the left hand side) and the optimized variant (on the right hand side) resulting from the application of the described optimization heuristics.

Following the optimization steps we sketched above, the optimizer applies various *standard operator simplifications* [Rit10, Ch. 4, p. 51ff] to derive the final plan shape. Since the optimized plan may be covered by a single (blue) tile, the SQL generator is able to condense the entire query plan into the single `SELECT·FROM·WHERE` block below (in Figure 5.1). In the following experiments we apply the above and various other optimizations to considerably simplify the query plans. We will see that the resulting queries can compete with hand-crafted SQL queries.

5.2 Benchmark

For our experimentation setup, we devise a sensible set of queries to assess the impact of this work. All of the queries may be used in the context of *Spree* [Spr], the Rails framework to construct E-Commerce applications that we already introduced in Section 2.2. As the underlying data model is very similar to the one of the TPC-H benchmark [Tpc], we slightly adapted the data generator to populate the tables of the *Spree* application we use in the experimentation setup. Consider Appendix A.1 for an illustration of the relevant *Spree* tables and associations between them.

Query	Characteristics
B_1	sorting, nested result, heavy materialization in the host language
B_2	filtering, huge intermediates, intermediate grouping, aggregation
B_3	filtering, sorting, huge intermediates, intermediate grouping, positional access
B_4	high selectivity, join-based, aggregation
B_5	filtering, aggregation, huge intermediates, deeply nested result, heavy materialization in the host language

Table 5.1: Notable characteristics of Queries B_1 through B_5.

We compare five queries, each of which emphasizing a different aspect of query formulation (Table 5.1). Each query comes in three forms, respectively focusing on a different programmer profile:

(i) a web application developer, who displays proficiency in ActiveRecord but lacks awareness of how the programs are evaluated on the database back-end,

Query	Result Types	# Queries
B_1	$[(\mathit{Str}, [(\mathit{Int}, \mathit{Str})])]$	2
B_2	Dec	1
B_3	[{`var_id`: *Int*, `size`: *Int*}]	1
B_4	Dec	1
B_5	[{`user`: *Int*, `orders`: [{`order`: *Int*, `suggestions`: [{`lineitem`: *Int*, `your_var`: {`id`: *Int*, `price`: *Dec*}, `sugg_var`: {`id`: *Int*, `price`: *Dec*}}], `saving`: *Dec*}]}]	3

Table 5.2: Result types of the SWITCH queries and number of emitted SQL queries. Consider Section 3.10.3, in which we detailed how the number of of queries is obtained by the result type.

(ii) a pure RUBY developer, who takes the complex and ordered data structures of RUBY for granted and uses SWITCH to express the queries, and

(iii) a database application developer, who composes a single SQL statement followed by a RUBY expression to construct the desired data structures in the host language.

In the following, we provide the respective queries to implement various tasks. For all SWITCH queries we listed the type and the emitted SQL queries that are fired against the database. The red code fragments mark query parts that are not considered to be database executable and thus are evaluated in the host language heap. The violet code fragments highlight the query parts that are literally adopted in the translated SQL code.

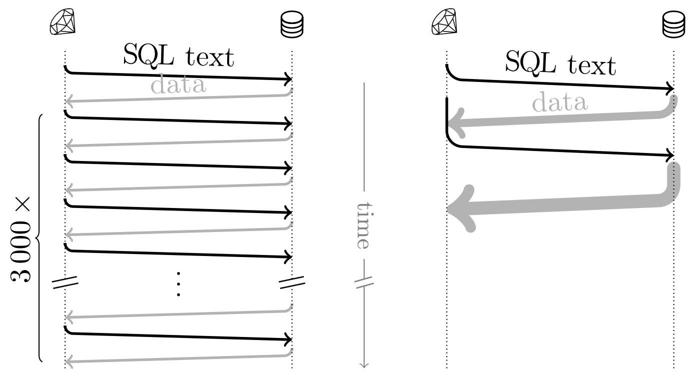
5.2.1 Drop-down List of Countries

“ *Creating a comprehensive list of countries and their associated states, alphabetically ordered and ready to be used for an online form.*

(B_1) ”

In Figure 5.3a we show the ACTIVERECORD variant of Query B_1. The emitted SQL queries are depicted in Figure 5.4a. As the first step, we gather all countries ordered by their name. Not surprisingly, this formulation leads to the single SQL query (line 1 in Figure 5.4a) that is executed on the database back-end. Following the materialization in the host language, we iterate through the countries and successively collect the associated states to construct the final result. For each country ACTIVERECORD triggers the collection of respective states by a separate query that employs the relationship between the tables `Countries` and `States`.

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. For example, against a TPC-H instance, the amount of generated SQL queries ranges from 30 queries (for scale factor 0.001) to 300,000 queries (for scale factor 10). ACTIVERECORD partially addresses this phenomenon, also known as the *1+n query problem*, but RAILS still suffers [Act, see `includes()`]. A flood of simple, look-alike queries (Figure 5.4a) keeps the back-end busy and the overall execution time is dominated by costly context switches between the RAILS and the database processes, which repeatedly exchange SQL text and tiny pieces of data (see Figure 5.2a).



(a) ACTIVERECORD. (b) SWITCH.

Figure 5.2: 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).

Figure 5.3b expresses the SWITCH variant of Query B_1. RUBY idioms such as `&:name` ≡ `{|x| x.name}` remain available to order the countries and states in line 1 and line 3, while the 1-to-n association to gather the states for each country is established in line 4. Because the final result is nested, the translation leads to exactly two SQL queries independent from the database instance size in Figure 5.4b. Note that the constructs enclosed by square brackets (`[...]`), such as `[id, name]` are regarded as a tuple by SWITCH and do not influence the nesting level. Here,

```
countries = Country.order("Countries.name")

countries.map { |c|
  [ c.name,
    c.states.order("States.name").
             map { |s| [s.id, s.name] } ]
}
```

(a) Variant of Query B_1 formulated from the angle of an ACTIVERECORD developer.

```
Countries.sort_by(&:name).map { |c|
  [ c.name,
    states.sort_by(&:name).
           select { |s| s.country_id == c.id }.
           map { |id,name| [id, name] } ]
}
```

(b) Variant of Query B_1 formulated from the angle of a RUBY purist using SWITCH.

```
query = <<-SQL
  SELECT c.name AS c_name, s.id AS s_id,
         s.name AS s_name
    FROM Countries c LEFT OUTER JOIN States s
           ON c.id = s.country_id
ORDER BY c.name, s.name;
SQL

ActiveRecord::Base.connection.select(query).
  group_by { |cs| cs["c_name"] }.
  map { |cname,states|
    [ cname,
      states.map { |s|
        [s["s_id"].to_i, s["s_name"]]
      }
    ]
  }
```

(c) Variant of Query B_1 formulated from the angle of a database application programmer.

Figure 5.3: Formulations of Query B_1 from the point of view of three different developers.

```
SELECT Countries.* FROM Countries ORDER BY Countries.name;
SELECT States.* FROM States WHERE (States.country_id = 1) ORDER BY States.name;
SELECT States.* FROM States WHERE (States.country_id = 2) ORDER BY States.name;
⋮
SELECT States.* FROM States WHERE (States.country_id = 2999) ORDER BY States.name;
SELECT States.* FROM States WHERE (States.country_id = 3000) ORDER BY States.name;
```

(a) SQL queries emitted by the ActiveRecord variant of Query B_1 (against a TPC-H instance of scale factor 0.1).

```
  SELECT 1 AS iter, a.name AS item1,
         ROW_NUMBER () OVER (ORDER BY a.id ASC) AS item2
    FROM countries AS a
ORDER BY a.name ASC;

WITH

cnt (iter, item1, item2) AS
(SELECT DENSE_RANK () OVER (ORDER BY b.id ASC) AS iter
        b.id AS item1, b.name AS item2,
   FROM countries AS b)

  SELECT c.iter AS iter, d.id AS item1,
         d.name AS item2
    FROM states AS d, cnt AS c
   WHERE d.country_id = c.item1
ORDER BY c.iter ASC, d.name ASC;
```

(b) SQL queries emitted by the Switch variant of Query B_1.

Figure 5.4: Comparison of the SQL queries emitted by ActiveRecord and Switch. Note that ActiveRecord generates queries that heavily depend on the underlying table size, whereas Switch always produces exactly the same queries regardless of the database instance size.

`ROW_NUMBER()` and `DENSE_RANK()` (in line 2 and line 9) correctly implement the surrogate values that permit us to assemble the result in the host language. Note how SWITCH expresses the 1-to-n association via an equi-join in lines 15 and 16, rather than a query for each single country. For countries without states no surrogate value is generated. This leads to empty inner list when the result is assembled in the host language.

Observe how the SWITCH formulation leads to a radically different interaction with the database back-end: regardless of the database instance size, exactly two SQL queries will be executed. Because all queries are independent, the execution may even overlap arbitrarily, to leverage the back-end's *buffer-pool-replacement strategy* (Figure 5.2b).

In Figure 5.3c we express Query B_1 in a single SQL query followed by a RUBY expression, which postprocesses the result on the host language heap. To consider countries that do not have associated states we use the `LEFT OUTER JOIN` construct. Ordered by the country names and state names, the query result is then divided into groups based on the country name in order to derive the desired shape.

5.2.2 Granting Discount to High-Volume Customers

“ *What would be the cost of granting a 10 percent discount to the open orders placed by all high-volume customers?* (B_2) ”

We already introduced the above question in Section 2.2.1 to illustrate the disadvantages arising when ACTIVERECORD is used to express more complex queries. The query in Figure 5.5a is a mere reprisal of the query we used in Figure 2.7 to implement the above question. In this case, dependent on the database instance size, the SQL translation generates huge `IN(...)` clauses that could easily overflow the back-ends parse buffer and fail to scale.

In Figure 5.5b we show the array-centric reformulation of Query B_2. Note how the style considerably differs from the ACTIVERECORD variant, which directly uses SQL query constructs in the formulation. 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 > highvol}`, 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). Observe also that we use several variables, such as `high_vols` or `open_orders`, which, as part of RUBY's evaluation process, are conveniently assembled into a single expression before the loop-lifted translation takes place.

After being optimized, the SWITCH expression in Figure 5.5b translates into the single SQL statement shown in Figure 5.6. Again, the purely relational approach we emphasize in this work pays out in full when the queries are presented to a relational back-end. DB2 recognizes further optimization potential and considerably simplifies the query into the SQL construct presented in Figure 5.7a. The Switch fragment that gathers the high-volume customers nests conveniently as an uncorrelated subquery in the `FROM` clause (see line 3 through line 5) followed by the predicate `oc.cnt > 10` (in line 11) that eventually filters all users with more than ten orders placed. Additionally, the query involves equi-joins on the SQL level to retrieve the remaining order columns (especially `order_id`) to resolve the 1-to-n association with the line items in line 10 and line 8 respectively.

In the SQL reformulation of Query B_2 (Figure 5.5c) we use an uncorrelated subquery that nests in an `IN(...)` clause to gather high-volume customers. The 1-to-n association to the line items is then established via an equi-join. A comparison of this formulation with the query in Figure 5.7a reveals their similarity and underlines the quality of the SQL queries emitted by Switch.

Interestingly, to evaluate these queries, DB2 constructs exactly the same execution plan depicted in Figure 5.7b (consider Table 5.3 for all relevant plan operators). To efficiently implement the retrieval of high-volume customers in one go with their orders a *merge-scan join*[1] is employed—the index access ensures that both input streams are sorted according to the `user_id`. Furthermore, in the right branch of the sort-merge join, the a B-Tree [BM02] index structure is leveraged to filter for open orders (`state = 'O'`). In the remaining plan, DB2 uses an index-nested-loop join to retrieve the line items based on their `order_id`. In the left branch, the optimizer recognizes that the outer table is poorly clustered and decides to sort it based on its join column (`id`) in the Orders table. In this way the number of read operations to access the inner table might be significantly reduced because they are more likely to be in the buffer pool already [Db2a]. The index structure on the inner table (right branch) allows an index-only access providing all columns (`price` and `quantity`) that are of interest in the following calculation without accessing the base table. The following grouping operator applies the `SUM()` aggregation function to calculate the arising costs.

5.2.3 Who Bought This Also Bought That

“*Offering the customer a list of items related to the current product by answering the question “Which items were purchased by customers who bought that item?”—only the three most popular items are delivered along with their popularity.* (B_3)”

The snippet in Figure 5.8a displays a typical ActiveRecord fragment that is intermingled with SQL code (violet code fragments) and fragments that are per-

[1]also known as *sort-merge join*

```
discount = 10.0/100
high_vol = 10

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_items).map(&:line_items).flatten
cost  = items.sum { |i| i.price * i.quantity } * discount
```

(a) Variant of Query B_2 formulated from the angle of an ActiveRecord developer.

```
discount = 10.0/100
high_vol = 10

high_vols = Orders.group_with(&: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_Items.select { |l| o.id == l.order_id }
        }.flatten
cost = items.map { |l| l.price * l.quantity }.sum * discount
```

(b) Variant of Query B_2 formulated from the angle of a Ruby purist using Switch.

```
ActiveRecord::Base.connection.select(<<-SQL).to_f
  SELECT SUM(l.price * li.quantity) * 0.1 AS cost
    FROM Orders o INNER JOIN Line_Items li
           ON o.id = li.order_id
   WHERE user_id IN (SELECT user_id
                       FROM Orders o
                    GROUP BY o.user_id
                      HAVING COUNT(*) > 10)
     AND state = 'O'
SQL
```

(c) Variant of Query B_2 formulated from the angle of a database application developer.

Figure 5.5: Formulations of Query B_2 from the point of view of three different developers.

```
WITH
usr (item12, item28) AS
  (SELECT a.user_id AS item12, COUNT (*) AS item28
     FROM orders AS a
 GROUP BY a.user_id),

oord (item45, item46, item47, iter48) AS
  (SELECT c.id AS item45, d.order_id AS item46,
          d.price * d.quantity AS item47,
          1 AS iter48
     FROM usr AS b, orders AS c, line_items AS d
    WHERE b.item12 = c.user_id
      AND 10 < b.item28
      AND c.state = '0'
      AND c.id = d.order_id),

total (iter48, item49) AS
  (SELECT e.iter48, SUM (e.item47) AS item49
     FROM oord AS e
 GROUP BY e.iter48)

SELECT 1 AS iter, (f.item49 * 0.1) AS item1
   FROM total AS f;
```

Figure 5.6: SQL encoding of Query B_2 directly emitted by SWITCH.

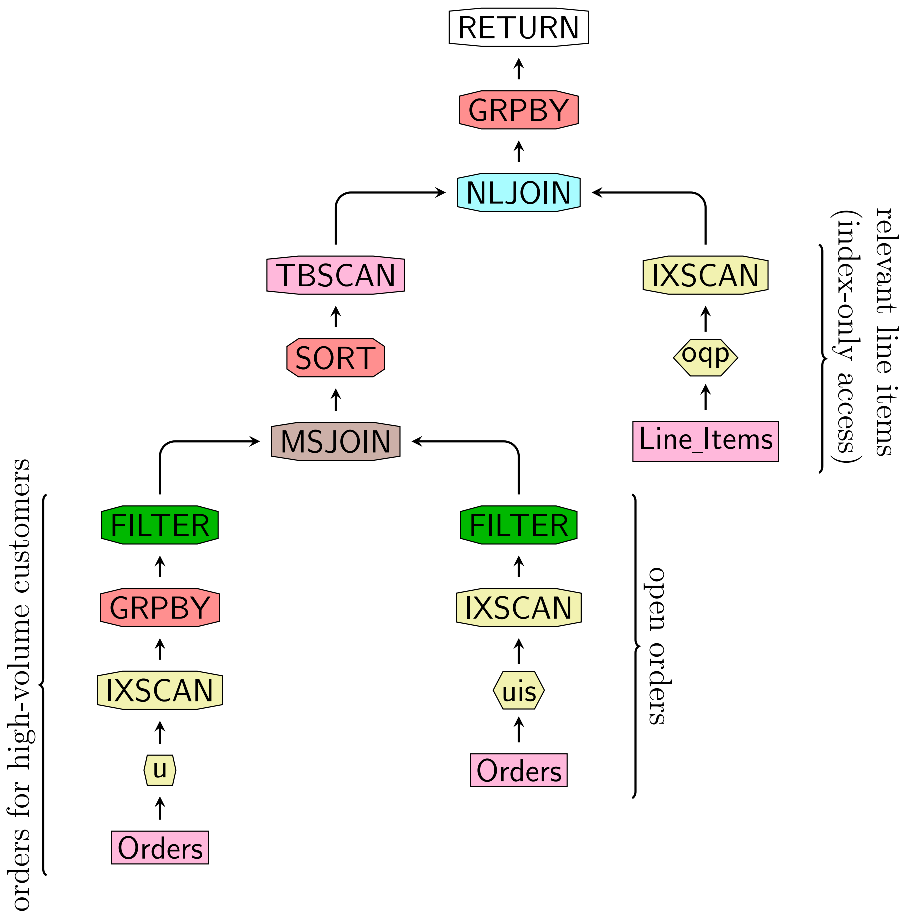
Operator	**Semantics**	**Operator**	**Semantics**
RETURN	Result row delivery	SORT	Sort rows (+ duplicate row elimination)
NLJOIN	Nested-loop join (left branch: outer table)	MSJOIN	Sort-merge join (left branch: outer table)
IXSCAN	B-tree scan	TBSCAN	Temporary table scan
u uis	Index access (Orders)	Table	Table access
oqp	Index access (Line_Items)		

u:user_id, i:id, s:state, o:order_id, q:quantity, p:price

Table 5.3: Relevant IBM DB2 plan operators.

```
SELECT 1 AS iter,
       SUM(li.price * li.quantity) * 0.1 AS item1
  FROM  (SELECT o2.user_id, COUNT(*) AS length
           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.length  > 10;
```

(a) SQL query emitted by SWITCH (Figure 5.6) after DB2 rewrite facilities took place. [Formatted for readability.]



(b) Execution plan constructed by DB2 to evaluate SWITCH formulation (Figure 5.5b) and the SQL formulation (Figure 5.5c) of Query B_2.

Figure 5.7: To evaluate Query B_2 on the back-end, DB2 devises exactly the same execution plan for the SWITCH variant and the SQL reformulation.

formed in the host language heap (red code fragments). We search products that have been bought together with a specific product variant (`variant = 286`). The `includes(:line_items)` construct prevents the 1+n query problem by generating a huge `IN(...)` list to gather the associated line items for each order. For a TPC-H instance of scale factor 0.1 a list with 150, 000 order ids needs to be assembled in the host language heap in order to populate the SQL query that is shipped to the back-end (Figure 5.9a in line 5). The third query filters and counts all variants that are actually ordered together with the relevant variant. Again, the query is dominated by a huge `IN(...)` list (Figure 5.9a in line 10) that fails to scale as it is applied to a TPC-H instance of scale factor 1.

The array-centric reformulation of Query B_3 is captured in Figure 5.8b. The list method `any?()` helps us to find all orders containing the considered variant. Switch also embraces user-defined methods written in an array-centric style: the singleton method `in_order()`, invoked in Figure 5.8b and defined as

```
class << line_items = Line_Items
  def in_order(o)
    select { |l| o.id == l.order_id }
  end
end
```

encapsulates the 1-to-n association between the *Spree* tables Orders and Line_Items, for example. The list function `take(3)` eventually delivers three items after they have been sorted according to their popularity.

The single SQL query (Figure 5.9b) that is generated by Switch presents a typical join graph, in which the innermost statement performs a series of joins followed by the blocking operators that involve sorting, grouping and aggregation. Observe that the positional access (`take(3)`) is elegantly expressed by `c.pos <= 3` in line 15. Column `c.pos` originates in a `ROW_NUMBER()` operator (line 3) that establishes a new logical order based on popularity.

The hand-crafted SQL in Figure 5.8c provides the most efficient variant to implement Query B_3. Following the single equi-join to establish the association between orders and line items grouping is applied to derive the popularity of the variants (`COUNT()`). By stating `FETCH FIRST 3 ROWS ONLY` we prepare the back-ends execution engine (and optimizer) to only gather the first three elements.

```
variant = 286
orders = Order.includes(:line_items).
                select { |o|
                  !o.line_items.select { |li|
                    li.variant_id == variant }.empty? }
varid_count = LineItem.
                where("order_id IN (:ordid) " +
                      "AND variant_id <> :varid",
                      { ordid : orders.map(&:id),
                        varid : variant }).
                count(:variant_id, group : "variant_id")
varid_count.sort_by { |vid,size| -size }.take(3)
```

(a) Variant of Query B_3 formulated from the angle of an ACTIVERECORD developer.

```
variant = 286
orders = Orders.select { |o|
           line_items.in_order(o).any? { |li|
             li.variant_id == variant } }

orders.map { |o| line_items.in_order(o) }.flatten.
       select { |li| li.variant_id != variant }.
       group_with { |v| v.variant_id }.
       map { |vid,li| { varid:vid, size:li.length } }.
       sort_by { |vid,size| -size }.take(3)
```

(b) Variant of Query B_3 formulated from the angle of a RUBY purist using SWITCH.

```
query = <<-SQL
    SELECT l1.variant_id, COUNT(DISTINCT l2.order_id) AS size
      FROM Line_Items l1 INNER JOIN Line_Items l2
           ON l1.order_id = l2.order_id
     WHERE l2.variant_id = 286
       AND l1.variant_id <> 286
  GROUP BY l1.variant_id
  ORDER BY size DESC
  FETCH FIRST 3 ROWS ONLY;
SQL

ActiveRecord::Base.connection.select(query).map { |v|
  { variant_id : v["variant_id"].to_i,
    size       : v["size"].to_i }
}
```

(c) Variant of Query B_3 formulated from the angle of database application developer.

Figure 5.8: Formulations of Query B_3 from the point of view of three different developers.

```
SELECT orders.* FROM orders;

SELECT line_items.*
  FROM line_items                          150,000 order ids
 WHERE (line_items.order_id IN (1,2,...,150000));

  SELECT COUNT(line_items.variant_id) AS count_variant_id,
         variant_id AS variant_id
    FROM line_items               600,572 order ids
   WHERE order_id IN (15431,17733,...,596292)
     AND variant_id <> 286
GROUP BY variant_id;
```

(a) SQL queries emitted by the ACTIVERECORD variant of Query B_3 (against a TPC-H instance of scale factor 0.1).

```
  SELECT 1 as ITER, c.vid AS item1, c.length AS item2
    FROM (SELECT b.vid, b.length,
                 ROW_NUMBER() OVER (ORDER BY -b.length) AS pos
            FROM (SELECT a.vid, COUNT(*) AS length
                    FROM (SELECT DISTINCT Q3.variant_id AS vid,
                                 Q1.ID AS ID
                            FROM Orders AS Q1,
                                 Line_Items AS Q2,
                                 Line_Items AS Q3
                           WHERE Q2.variant_id = 286
                             AND Q1.ID = Q2.order_id
                             AND Q1.ID = Q3.order_id
                             AND Q3.variant_id <> 286) AS a
                GROUP BY a.vid) AS b) AS c
   WHERE c.pos <= 3
ORDER BY c.pos;
```

(b) SQL encoding of Query B_3 emitted by SWITCH after having been rewritten by DB2 [Formatted for readability.]

Figure 5.9: SQL queries produced by the ACTIVERECORD and SWITCH variant of Query B_3.

5.2.4 Checkout and Cheapest Variants

Simulation of the last step in the checkout process in which the total cost (including taxes) of the purchased products is presented to the customer. (B_4)

The checkout process is the linchpin in every E-Commerce application. The above query calculates the total cost (taxes included) of a specific order. Roughly, Query B_4 has to perform joins between six tables followed by an aggregation function (`SUM()`) to faithfully derive the proper tax rate associated with a product. The query operates in a realistic scenario: the order (`id` = `807`) contains exactly 7 line items for all instances and is, due to its high selectivity[2], made for a relational database system.

For the SWITCH variant and the SQL variant (Figure 5.10b and Figure 5.10c respectively), DB2's cost-based optimizer recognizes high selectivity due to its distribution statistics and decides to bring corresponding joins forward to prepare an efficient execution plan. Note that we captured the associations between the various tables by singleton methods as shown in the previous section.

The ACTIVERECORD variant solely relies on the favorable formulation, captured in Figure 5.10a. The finder method `find_by_id()` guarantees fast access (supported by an index) to a specific order to subsequently gather the associated line items. The `includes()` construct in turn establishes the required associations via `IN(...)` lists with each list containing exactly 7 items. Note that the slightest difference in the formulation, for example *perform the filter for a specific order last in the query*, leads ACTIVERECORD to yield a completely different set of SQL statements. Inherently, the query formulation in ACTIVERECORD rigidly prescribes the execution order of the expressions.

SWITCH, however, remains stable under such circumstances. The single SQL query derived for Query B_4 leaves it to back-end's optimizer to possibly reinvent the evaluation strategy in order to provide an efficient execution plan that introduces highly-selective operators early on.

We provide the chance to make substantial savings by enabling the customer to review the cheapest variants for all chosen products of their orders. We consider only orders for which the customers have not yet completed the checkout. (B_5)

The last query we test addresses the not yet completed orders of all customers, so that they may compare the chosen items of their orders with the cheapest variants that can be found in the product catalog of the web-store. The result is a

[2]for scale factor 10 the `Line_Items` table contains nearly 60 million entries

```
order_id = 807

order = Order.find_by_id(order_id)

line_items = order.line_items.includes(
               {:variant => {
                  :product => { :tax_category => :tax_rates} }})

line_items.map { |li|
  tax_rate = li.variant.product.tax_category.
                tax_rate.amount
  li.price * li.quantity * (tax_rate + 1)
}.sum
```

(a) Variant of Query B_4 formulated from the angle of an ActiveRecord developer.

```
order_id = 807

Orders.by_id(order_id).line_items.map { |li|
  tax_rate = li.variant.product.tax_category.
                tax_rate.amount
  li.price * li.quantity * (tax_rate + 1) }.sum
```

(b) Variant of Query B_4 formulated from the angle of a Ruby purist using Switch.

```
ActiveRecord::Base.connection.select(<<-SQL).to_f
  SELECT SUM(li.price * li.quantity * (tr.amount + 1))
    FROM Orders o, Line_Items li, Variants v, Products p,
         Tax_Rates tr, Tax_Categories tc
   WHERE o.id = 807
     AND li.order_id = o.id
     AND li.variant_id = v.id
     AND v.product_id = p.id
     AND p.tax_category_id = tc.id
     AND tc.id = tr.tax_category_id;
SQL
```

(c) Variant of Query B_4 formulated from the angle of database application developer.

Figure 5.10: Formulations of Query B_4 from the point of view of three different developers.

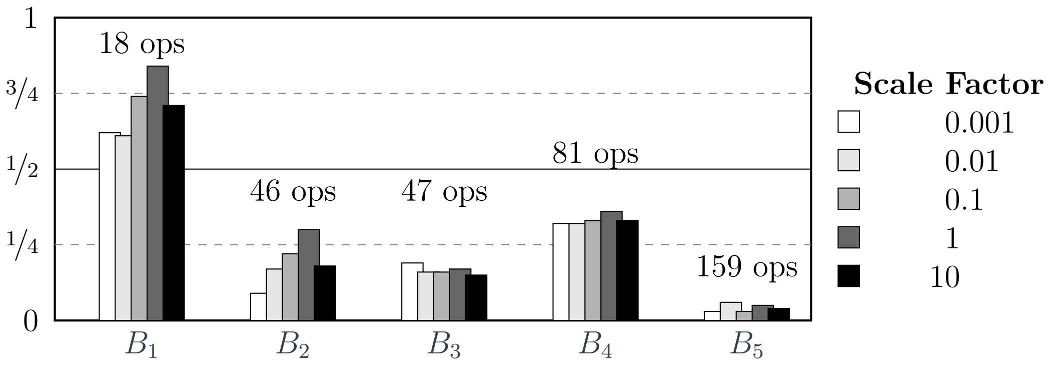


Figure 5.11: Relative Execution time of the standard SQL code generation in comparison to the simple approach for various database instance sizes. The numbers atop of the bars show the number of operators (ops) participating in the loop-lifted query plan of each query as a metric for its structural complexity.

list of records, in which each record comprises a user together with his orders. For each line item in an order the identifier of the chosen product is listed along with the cheapest variant. Additionally for each order the cost that could be saved by choosing the cheapest variant is shown.

Because this query is the most complex one in our setting, we decided to defer the formulation of all variants of Query B_5 to Appendix A.2. The formulations are accompanied by a more detailed discussion, in which we explain the interesting aspects of this query.

5.3 SQL Code Generation

During our experiments we quickly realized that the simple SQL translation scheme approach (see Table 4.1 on page 110) is not suitable to keep pace with handcrafted SQL code. The simplistic approach, which maps each operator in the loop-lifted query plan directly to a SQL query, devastated every query optimizer we had on our workbench. Even small query plans lead to sizable SQL queries, in which nesting and repeated sub-queries are pervasive (Figure 4.1 on page 112).

For the SQL query that the simple SQL code generator assembles for Query B_2, DB2's query optimizer arranged the execution plan in Figure 5.12. In comparison to the execution plan that we derived by our standard SQL code generation (see Figure 5.7b on page 146) this plan is noticeably more complex and contains two additional joins that could be avoided. Furthermore, due to the complex SQL queries that are fed to DB2, the query optimizer is seemingly not able to properly identify the sharing opportunities offered by the loop-lifted compilation.

The plan fragments we covered with gray tiles are equivalent. These repeated sub-plans coalesce into the immediately following join operator (`MSJOIN`), which merges the data based on the primary-key column id in the Orders table. The query optimizer recognizes the uniqueness of the join-attributes, indicated by the early-out flag [Ear] (*). This additional join could have been avoided if the query optimizer had been aware of the equivalence of both sub-plans, as it is the case for the plan derived by the standard SQL generator.

The situation turns out to be even more challenging for the query optimizer if the query plans become more complex (in terms of participating operators), or if rownumber operators (`ROW_NUMBER()`) or rank operators (`DENSE_RANK()`) participate in the evaluation. In this case, the back-end fails completely to arrange efficient execution plans. In Figure 5.11 we illustrate the impact of the standard SQL code generator in comparison to the simple approach.

5.4 Quantitative Assessment

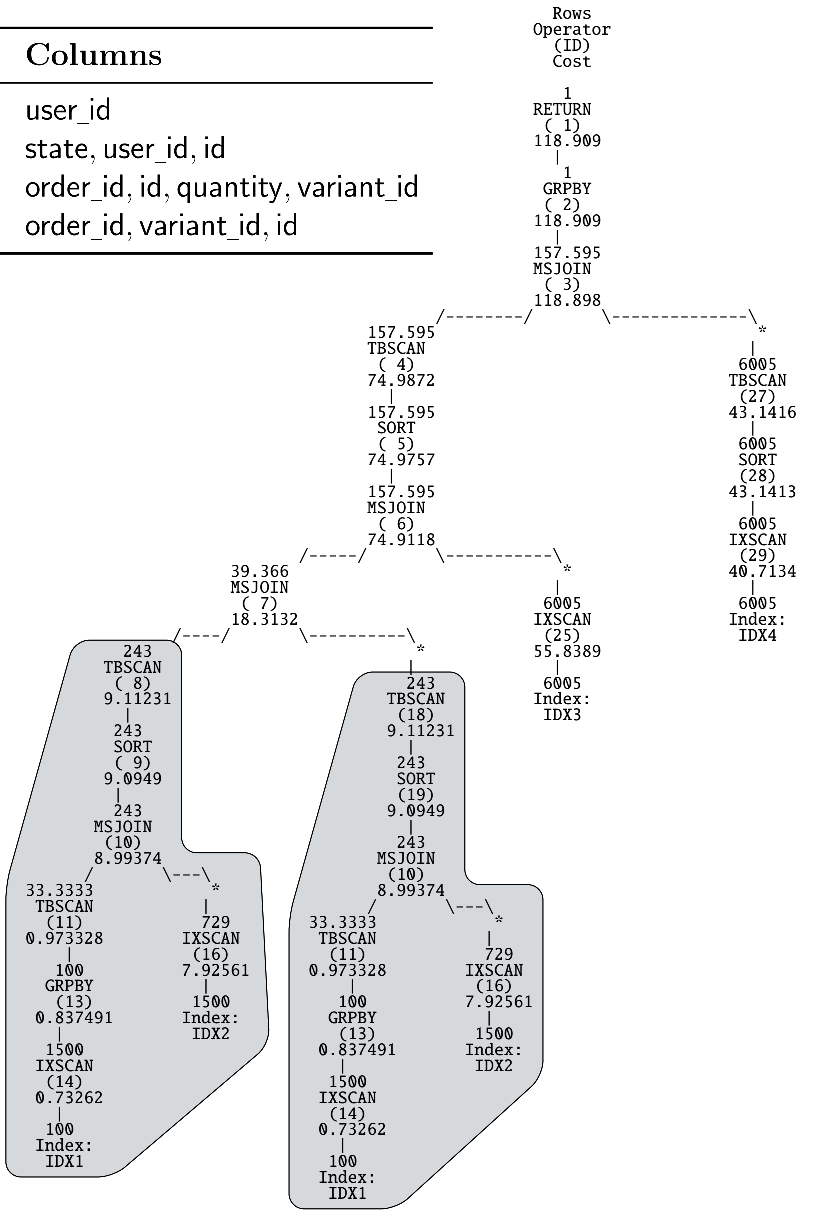
In this section, we focus on the runtime characteristics of the query emitted by Switch. We compare the runtime effects of Switch with the ActiveRecord variants and SQL variants we detailed in the previous sections and examine how DB2 copes with the different query instances.

All experiments were performed on a Sun Fire X4275 server. The host runs Linux v2.6 with DB2 Viper v9.7 and is set up with two Intel Xeon™ processor units X5570 (2.95 GHz quad core), 72 GB main-memory, and a three SCSI disk with 6 TB of storage in total.

For all queries, we measure the complete round-trip time that possibly comprises (a) the loop-lifted translation, (b) query optimization, (c) SQL generation (d) deriving intermediate results in the host language, (e) query execution on the back-end, (f) collecting the results from the back-end, and (g) deriving the desired result shape in the host language (consider Table 5.4 for the detailed steps involved in the different approaches).

For all queries we ran DB2's *design advisor* [Db2b] that autonomously chooses indexes and auxiliary structures that meet the demands of the workload. Following the reorganization of all participating tables into unfragmented, physically contiguous pages (`REORG` [Db2c]), we instructed DB2 to gather detailed distribution statistics on all tables via the `RUNSTATS` command [Db2d]. All SQL queries compiled by the back-end with optimization level 9.

Index	Table	Columns
IDX1	Orders	user_id
IDX2	Orders	state, user_id, id
IDX3	Line_Items	order_id, id, quantity, variant_id
IDX4	Line_Items	order_id, variant_id, id



```
                                                      Rows
                                                    Operator
                                                      (ID)
                                                      Cost

                                                       1
                                                    RETURN
                                                     ( 1)
                                                    118.909
                                                      |
                                                       1
                                                     GRPBY
                                                     ( 2)
                                                    118.909
                                                      |
                                                    157.595
                                                    MSJOIN
                                                     ( 3)
                                                    118.898
                                          /--------/       \--------------\
                                   157.595                                 *
                                   TBSCAN                                  |
                                    ( 4)                                 6005
                                   74.9872                              TBSCAN
                                     |                                  (27)
                                   157.595                              43.1416
                                    SORT                                   |
                                    ( 5)                                 6005
                                   74.9757                               SORT
                                     |                                   (28)
                                   157.595                              43.1413
                                   MSJOIN                                  |
                                    ( 6)                                 6005
                                   74.9118                              IXSCAN
                            /-----/       \-----------\                  (29)
                     39.366                            *                40.7134
                     MSJOIN                           |                    |
                      ( 7)                         6005                  6005
                     18.3132                    IXSCAN                  Index:
               /----/       \----------\         (25)                    IDX4
          243                            *           55.8389
        TBSCAN                          |              |
         ( 8)                          243           6005
        9.11231                      TBSCAN         Index:
          |                           (18)           IDX3
         243                         9.11231
         SORT                          |
         ( 9)                         243
        9.0949                        SORT
         |                            (19)
         243                         9.0949
       MSJOIN                          |
        (10)                          243
       8.99374                      MSJOIN
      /       \---\                  (10)
33.3333            *                8.99374
 TBSCAN           |                /       \---\
  (11)            729         33.3333           *
0.973328        IXSCAN        TBSCAN           |
    |            (16)          (11)            729
   100          7.92561      0.973328        IXSCAN
  GRPBY           |              |            (16)
   (13)         1500            100          7.92561
 0.837491       Index:         GRPBY           |
   |             IDX2           (13)          1500
  1500                        0.837491       Index:
 IXSCAN                         |             IDX2
  (14)                         1500
 0.73262                      IXSCAN
   |                           (14)
  100                         0.73262
 Index:                         |
  IDX1                         100
                              Index:
                               IDX1
```

Figure 5.12: Execution plan that DB2 arranged for the SQL query of Query B_2 translated by the simple translation scheme for a database instance size of scale factor 1. Consider the repeated sub-queries (covered by gray tiles) that have not been properly recognized by the query optimizer.

			ACTIVERECORD	SWITCH	*SQL*	
Compilation	(a)	loop-lifted translation	–	✓	–	
	(b)	query optimization	–	✓	–	
	(c)	SQL generation	✓	✓	–	
	(d)	intermediate results	✓	–	–	Query Execution
	(e)	query execution	✓	✓	✓	
Materialization	(f)	collecting query results	✓	✓	✓	
	(g)	derive the result shape	✓	✓	✓	

Table 5.4: The above table details the steps affecting the evaluation time for the different approaches.

5.5 Queries on DB2

We measured the evaluation times for several scale factors (sf 0.001 through sf 10) of five queries (B_1, B_2, B_3, B_4, and B_5) for all three variants we proposed in the previous section (15 queries in total). We executed each query ten times and reported the average wall-clock evaluation time in milliseconds. Table 5.5 lists the results of these measurements. To get an initial impression of the performance of the SWITCH queries consider Figure 5.13, in which we put SWITCH into perspective against its contenders.

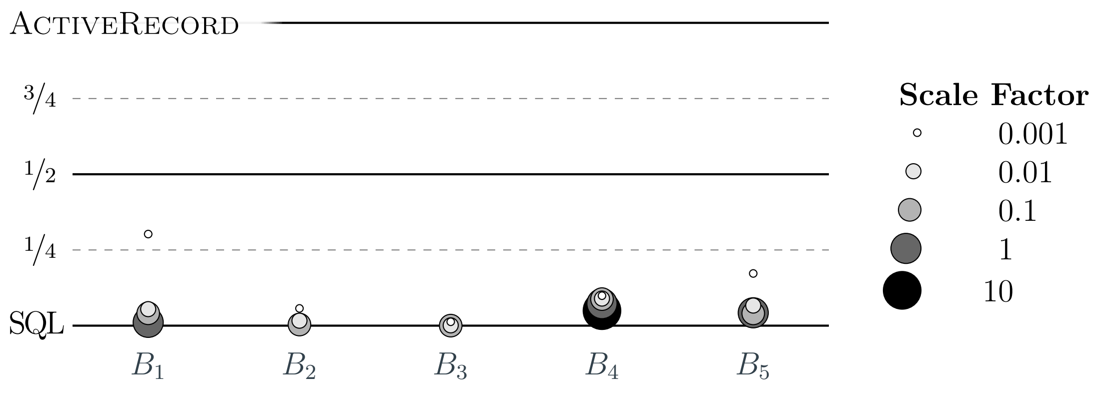


Figure 5.13: Relative execution times of Queries B_1 through B_5

Query	TPC-H Scale	⏱ (ms)				
		ACTIVERECORD		**SWITCH**		SQL
B_1	0.001	121	$\overset{\approx 3}{>}$	**38**	$\overset{\approx 19}{>}$	2
	0.01	871	$\overset{\approx 11}{>}$	**74**	$\overset{\approx 2.6}{>}$	28
	0.1	10,812	$\overset{\approx 18}{>}$	**592**	$\overset{\approx 3.7}{>}$	156
	1	368,988	$\overset{\approx 52}{>}$	**7,016**	$\overset{\approx 2.3}{>}$	2,996
	10	DNF	$\overset{?}{>}$	**63,795**	$\overset{\approx 2.0}{>}$	31,163
B_2	0.001	814	$\overset{\approx 15}{>}$	**53**	$\overset{\approx 7.5}{>}$	7
	0.01	8,112	$\overset{\approx 46}{>}$	**176**	$\overset{4.0}{>}$	44
	0.1	81,290	$\overset{\approx 91}{>}$	**893**	$\overset{\approx 1.5}{>}$	595
	1	Fails	$\overset{?}{>}$	**11,910**	$\overset{\approx 1}{>}$	12,162
	10	Fails	$\overset{?}{>}$	**128,016**	$\overset{\approx 1}{>}$	119,794
B_3	0.001	885	$\overset{\approx 26}{>}$	**34**	$\overset{\approx 1.4}{>}$	23
	0.01	8,923	$\overset{\approx 287}{>}$	**31**	$\overset{\approx 1.4}{>}$	21
	0.1	92,938	$\overset{\approx 2,655}{>}$	**35**	$\overset{\approx 1.4}{>}$	24
	1	Fails	$\overset{?}{>}$	**35**	$\overset{\approx 1.6}{>}$	21
	10	Fails	$\overset{?}{>}$	**37**	$\overset{\approx 1.6}{>}$	22
B_4	0.001	101	$\overset{\approx 5}{>}$	**19**	$\overset{\approx 1.9}{>}$	10
	0.01	124	$\overset{\approx 5}{>}$	**21**	$\overset{\approx 1.9}{>}$	11
	0.1	122	$\overset{\approx 6}{>}$	**18**	$\overset{\approx 2.2}{>}$	8
	1	111	$\overset{\approx 5}{>}$	**22**	$\overset{\approx 1.4}{>}$	15
	10	115	$\overset{\approx 6}{>}$	**19**	$\overset{\approx 1.3}{>}$	14
B_5	0.001	259	$\overset{\approx 4}{>}$	**67**	$\overset{\approx 2.5}{>}$	27
	0.01	1,196	$\overset{\approx 5}{>}$	**211**	$\overset{\approx 1.5}{>}$	141
	0.1	4,662	$\overset{\approx 5}{>}$	**932**	$\overset{\approx 1.2}{>}$	774
	1	52,148	$\overset{\approx 3}{>}$	**15,066**	$\overset{\approx 1.1}{>}$	13,421
	10	DNF	$\overset{?}{>}$	**603,406**	$\overset{\approx 1.1}{>}$	546,783

Table 5.5: Comparison between DB2's evaluation times (in milliseconds) for Queries B_1 through B_5. We tested three different variants (ACTIVERECORD, SWITCH, and SQL) for the scale factors 0.001–10. DNF marks the queries that did not finish within 10^6 milliseconds, Fails indicates that a query overflows DB2's parse buffer and thus fails to scale.

	B_1	B_2	B_3	B_4	B_5
Compilation Time (ms)	12	33	22	11	41

Table 5.6: Compilation time (in milliseconds) required to translate a SWITCH expression into a database executable SQL query. We ran the compilation ten times and reported the average.

5.5.1 Switch versus ActiveRecord

No SWITCH query performs more slowly than its ACTIVERECORD variant. Particularly, for Query B_1, Query B_2, Query B_3, and Query B_5 the experiments indicate a drastic reduction of evaluation time that ranges up to several orders of magnitude (see $\overset{x}{>}$). As we observed in the previous sections, these queries generate a considerable amount of intermediates that, due to ACTIVERECORD's partial execution strategy, lead to either a flood of look-alike queries or huge `IN(...)` clauses.

For the SWITCH variants, the positive effects emerge especially for larger TPC-H instances. For a database instance size of scale factor 10, the ACTIVERECORD variants for Query B_1 and Query B_5 do not even manage to deliver results within the given time frame. For a TPC-H instance of scale factor 1 or larger, ACTIVERECORD overflows DB2's parse buffer with SQL text of more than 10^7 Bytes (due to huge `IN(...)` clauses) and thus fails to scale (Query B_2 and Query B_3).

Interestingly, for Query B_3 the ACTIVERECORD variant shows a different scale behavior compared to the SWITCH variant and the SQL variant (see Figure 5.14). This is attributed to the huge intermediate results that ACTIVERECORD materializes twice in the host language (see Figure 5.8a on page 148) in order to construct the SQL queries. Here, the fragments executed in the host language range from a doubly nested `select()` method, and a `map()` operation (entailing several iterations over all available orders) to a `sort_by()` operation directly followed by `take()` to deliver the most popular items. For the SWITCH variant and the SQL variant the single SQL query benefits from the sophisticated optimization facilities and data-access mechanisms (such as index structures and several join variants) provided by DB2.

Because we only calculate the checkout costs for a single order, for Query B_4 all three variants benefit from the high selectivity. For the SWITCH variants and SQL variants, based on the distribution statistics, the optimizer prepares an execution plan that performs high-selectivity operations early on. The ACTIVERECORD variant, however, has to rely on a favorable formulation to recognize high selectivity early in order to keep the generated queries (and `IN(...)` clauses) to a minimum.

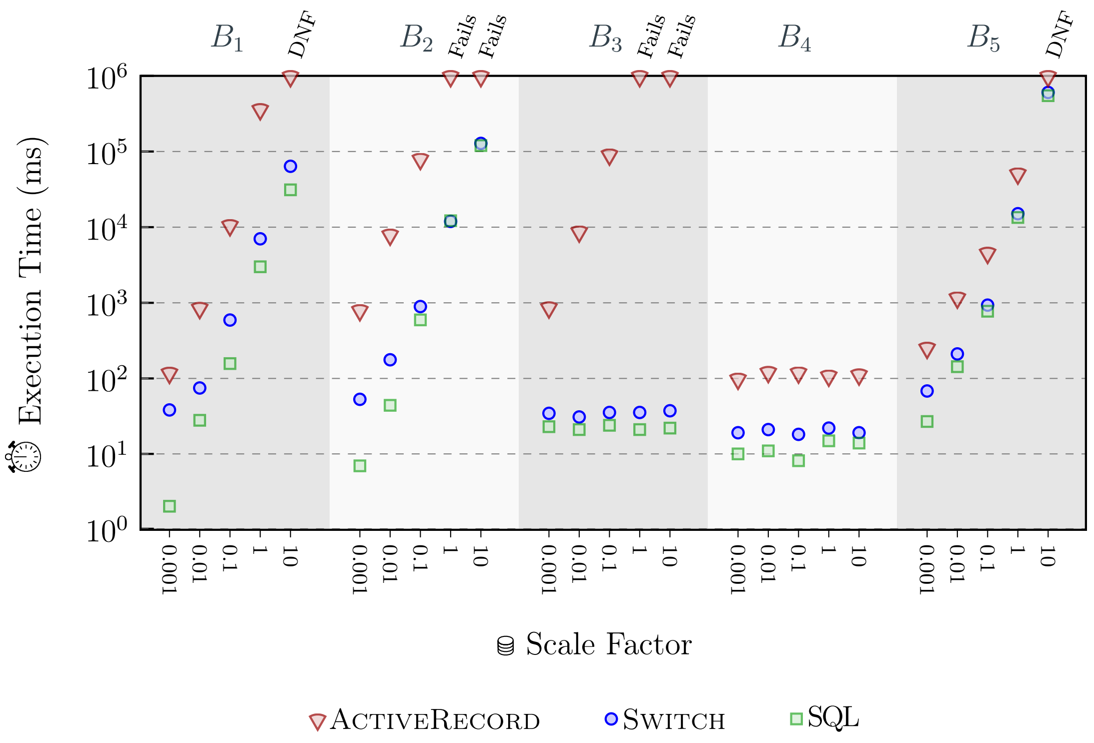


Figure 5.14: Measurements for Queries B_1, B_2, B_3, B_4, and B_5: the table reports the execution time in milliseconds averaged over 10 runs.

5.5.2 Switch versus SQL

The comparison between SWITCH and SQL demonstrates that the loop-lifted translations can compete with the SQL reformulations. All queries formulated using SWITCH produce the same graph pattern as their SQL counterparts (see Figure 5.14). The previous observations showed that both formulations result in a very similar (or even equivalent) execution plan that is eventually executed on the back-end.

Two outliers are found for Query B_1 and Query B_5 (scale factor 0.001). For these SWITCH queries, the evaluation time is largely dominated by the compilation time required to bring a SWITCH expression into a database executable format (see Table 5.6). For larger TPC-H instances, however, the performance gap shrinks significantly.

Chapter 6

Wrap-Up

In 1987, Atkinson and Buneman observed that "Databases and programming languages have almost developed independently of one another for the last twenty years" [AB87]. Consequently, they emphasized the need for a *uniform language* that smoothly integrates with modern programming languages and exploits the techniques and facilities that relational database systems can offer for processing considerable amounts of data. More than twenty years later, the problem of diverging database and programming-language domains still persists and this gap is nowhere more apparent than in nowadays web-programming frameworks.

Whereas programming languages support order and nesting as well as data abstraction, relational database systems still operate on flat relational tables. Most web developers are confronted with both paradigms and additionally bear the burden of mediating between them. In this two-paradigms approach, developers must repeatedly undergo the mental shift from the host language to SQL (and vice versa).

With Rails, the Ruby community partially addressed this problem. ActiveRecord enables developers to specify simple queries directly in the host language and bypass clumsy JDBC-like wrapper libraries. The query facilities provided by ActiveRecord, however, leave much to be desired. Ruby code is still sprinkled with literal SQL text fragments. The concatenative semantics of query methods rather reminds one of SQL than of Ruby. In the last years, smooth integration of database systems with programming languages has been receiving renewed attention. Relevant work from academia and industry have been discussed in Section 3.16.

6.1 Summary

In this thesis we made an attempt to cure the problems ACTIVERECORD suffers from. We picked up the ideas of loop lifting and presented a path towards query integration with which there is no syntactic or stylistic difference between RUBY programs that operate over in-memory arrays objects or database-resident tables, even if these programs rely on array order and nesting. Our development efforts assembled into SWITCH, a full-fledged compiler that translates RUBY expressions all the way down to SQL. SWITCH's built-in compiler and SQL generator guarantees to emit few queries, addressing long standing performance problems that trace back to RAILS' ACTIVERECORD database binding. In the following three sections, we will outline the steps that led to a successful implementation.

6.1.1 Query Integration into Ruby

We capture the program's structure in preparation for query generation. Developers may continue to use the host language's versatile family of array operations (found in the RUBY modules `Array` and `Enumerable`). The set of supported operations includes

```
concat [·] first last take drop reverse length flatten sum avg min
max member? uniq zip unzip map select flat_map all? any? take_while
drop_while count sort_by min_by max_by group_with partition max_by
min_by ,
```

as well as arithmetics, comparisons, and Boolean operations. We respect array order and support computation over nested arrays that may occur in both intermediates and final results (which naturally arise with `group_with` or `partition`, for example). Futhermore, we encourage programming with iterative first-order constructs—such as `map` and `select`. The deep-embedding approach, in which query expressions are collected during program execution, does not impede developers to use RUBY-specific idioms and to define user-specific methods in an array-centric style.

6.1.2 A Relational Portrayal of SWITCH

We expanded on the loop-lifting techniques and provided a set of rules to directly transfer source-level constructs into their tabular representation. In this context, a variant of the classical relational algebra, which assumes the role of an intermediate language, emphasizes the relational pureness and displays sufficient flexibility to reflect the capabilities of modern relational database management systems.

Loop lifting enables us to capture the independent evaluation of the iterative side-effect-free RUBY constructs and to turn them into a fully *set-oriented* representation, which blends well with SQL-centric back-ends. The tabular encodings with the schema $\langle \mathsf{iter}, \mathsf{pos}, \overline{\mathsf{item}} \rangle$ that are created during the compilation carry two additional columns—iter and pos —in order to correctly express iteration and to preserve the logical order between list items. The $\overline{\mathsf{item}}$ columns (which denotes a shortcut for the sequence $\mathsf{item}_1, \ldots, \mathsf{item}_\mathsf{n}$) accommodate the actual data.

In contrast to ACTIVERECORD or LINQ, loop lifting does not tend to produce avalanches of simple look-alike queries, which lead to an inefficient repetitive query-after-query mode of execution. The number of query plans we emit only dependents on the nesting depth of the result type. To account nesting, a result table carries a column, populated with surrogate values that make it possible to refer to nested values.

6.1.3 SQL Code Generation

The query plans emitted by the loop-lifted compilation scheme are fed to the SQL:1999 code generator. The code-generation approach we proposed applies a greedy strategy to partition the query plans into separate *tiles*. Each tile might consist of several relational primitives that may be faithfully condensed into a single SQL statement. The sequence of SQL statements we obtain are then assembled to form a SQL common table expression (`WITH` ...), which collectively implements the semantics of an input plan.

The experiments we conducted demonstrate that deep-embedding does not necessarily contradict competitive query performance. In fact, the generated queries can often contend with handcrafted SQL code and may outperform the ACTIVERECORD variants by several orders of magnitude. The generated SQL queries leave it to the back-end to choose for an efficient execution plan and thus can benefit from the large body of optimization techniques integrated in such systems.

6.2 Ongoing and Future Work

Whereas we only focused on the marriage between RUBY and relational back-ends, we are confident that the techniques developed in this work may be well applicable to other host languages and back-ends.

6.2.1 More Host Languages

Apart from RAILS, other popular frameworks are in use. With DJANGO [Dja] the PYTHON community contributed a similar framework that aims to tame the complexity of web programming. In this context, the techniques we developed in this work may be adopted to bring substantial amounts of PYTHON into the reach of SQL-centric back-ends. Like RUBY, PYTHON is a fully interpreted language that features a dynamic type system. PYTHON, however, inherently distinguishes between tuples and lists, and natively supports *list comprehensions* as a syntactic construct in form of

$$[e_{\text{body}}\ \texttt{for}\ v\ \texttt{in}\ e\ \texttt{if}\ e_{\text{cond}}]$$

along with a rich set of list functions.

6.2.2 Ogling at Map-Reduce Frameworks

When Dean and Ghemawat pioneered the Map-Reduce programming model in their seminal work [DG08], they shifted the spotlight to *distributed computing* as a viable alternative to RDBMS for processing and analyzing large-scale data. The idea is simple: Clusters of heterogeneous computers collectively work on a single problem in parallel. The underlying programming model was heavily inspired by the *map* and *reduce*[1] functions commonly used in functional programming, although they slightly differ from their original forms.

Although the programming model was often criticized [SA+10] (partly for its low-level interface), Apache™ launched the open-source project HADOOP® [Had] that has been established as a thriving playground for several Map-Reduce-based languages [PD+05; OR+08]. Particularly HIVE [TS+09] is of interest in this context, since it enables developers to interface HADOOP® via HQL—a declarative language similar to SQL. Much like in SWITCH, a variant of the relational algebra assumes the role of an intermediate language. This intermediate language is then directly translated into a sequence of Map-Reduce jobs that jointly implement the semantics of the underlying HQL expression. A translation into HQL (or its relational intermediate language) could bring SWITCH within touching distance of a promising new back-end. Similar work has been done by Microsoft [YI+08] in order to feature LINQ on their own Map-Reduce framework [IB+07].

An efficient translation upon vector-based in-memory database systems—such as MonetDB/X100 [BZN05]—may add another challenge and is actively researched at the Eberhard Karls Unversität Tübingen.

6.2.3 Alternatives to Loop Lifting

In Section 3.16, we already mentioned that the flattening transformation may come into question for a replacement of loop lifting. An approach that lets participate the flattening transformation in the compilation scheme would add another compiler stage that could significantly facilitate the translation into relational algebra

[1]also known as *fold*

by applying a series of source-level transformations to the input language. This path is currently followed at the Eberhard Karls Unversität Tübingen.

6.2.4 Proving Switch

We already began to set up a logical framework for formal study and analysis of Switch. For this purpose, we chose the Coq proof assistant [Coq] and its powerful inductive constructions to rigorously investigate Switch peculiarities.

So far, our efforts include the inductive formulation of the core constructs, the type system, and the coercion rules we proposed for Switch. We also defined a formal variant of the operational behavior of Switch constructs by means of a *small-step semantics* [Pie02, ch. 3, p. 32ff]. We are confident that these definitions open the perspective to formally prove the soundness of Switch.

Appendix A

Assessment

A.1 Associations between Spree Models

The below picture depicts the relevant fragment of *Spree* models and associated relationships that participate in the evaluation of Queries B_1 through B_5.

In Table A.1 we list the amount of tuples that reside in the database tables for all database instance sizes we considered in our experiments.

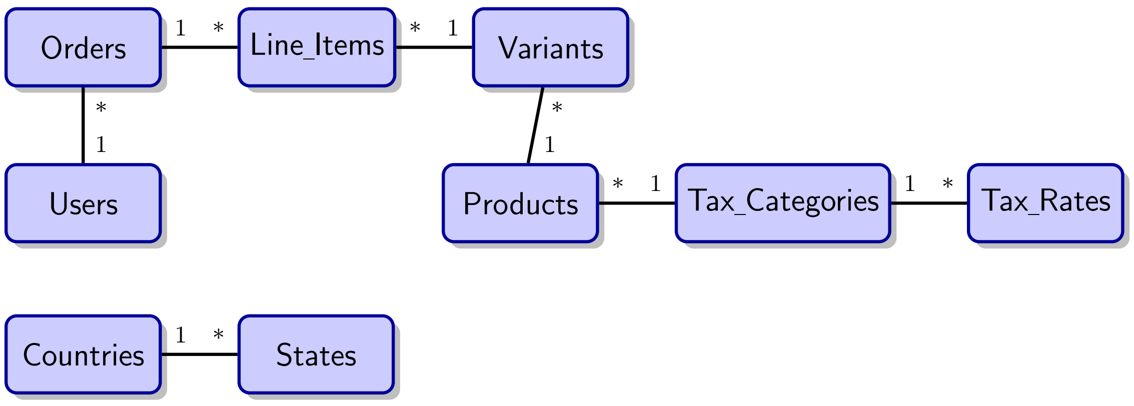


Figure A.1: Relevant *Spree* models and associations

Tables	TPC-H Scale (# Tuples)				
	0.001	0.01	0.1	1	10
Orders	$1,5 \times 10^3$	$1,5 \times 10^4$	$1,5 \times 10^5$	$1,5 \times 10^6$	$1,5 \times 10^7$
Line_Items	6×10^3	6×10^4	6×10^5	6×10^6	6×10^7
Variants	10^3	10^4	10^5	10^6	10^7
Products	200	2×10^3	2×10^4	2×10^5	2×10^6
Tax_Categories	10	10^2	10^3	10^4	10^5
Tax_Rates	20	2×10^2	2×10^3	2×10^4	2×10^5
Countries	30	300	3×10^3	3×10^4	3×10^5
States	90	900	9×10^3	9×10^4	9×10^5

Table A.1: Amount of tuples that reside in the tables for various TPC-H scale factors.

A.2 Cheapest Variants

Here we resume the discussion on Query B_5 from Section 5.2.4. Regarding its structure this query is the most complex one in our setting and combines nearly all techniques we used in the previous queries.

The ActiveRecord variant shown in Figure A.2 entails considerable post-processing in the host-language heap that dominates the overall evaluation time (consider the red code fragments). For a TPC-H instance of scale factor 1 nearly $1,500$ orders are materialized and left to the host language to group them by their `user_id` (see line 3).

For each order, ActiveRecord generates a separate SQL query to gather the associated line items (see line 7), leading to the typical abundance of context switches between the host language and the SQL back-end.

But that is not all: Because each order is connected to 4 line items (on average), ActiveRecord causes $6,000$ simple SQL queries to be fired against the database in order to get the proper variant (see line 8). Another $6,000$ queries are placed in order to get the variants with the same `product_id`, followed by the derivation of the cheapest variants in the host language (see lines 9 through 10).

The Switch variant of Query B_5 is shown in Figure A.3. Since Switch only generates a separate SQL query for each nesting level, three queries are assembled to outsource substantial work to the relational back-end.

```
orders = Orders.where("checkout_complete = ?", 1)

orders.group_by(&:user_id).map { |u,os|
  { user   : u,
    orders :
      os.map { |o|
         suggestions = o.line_items.map { |li|
             yvar = li.variant
             svar = Variants.where("product_id = ?",
                                   yvar.product.id).min_by(&:price)
             { lineitem : li.id,
               your_var : { id : yvar.id, price : yvar.price },
               sugg_var : { id : svar.id, price : svar.price } }
         }

         { order       : o.id,
           suggestions : suggestions,
           saving      : suggestions.sum { |v|
                           v[:your_var][:price] - v[:sugg_var][:price]
                         } }
      } }
}
```

Figure A.2: Variant of Query B_5 formulated from the angle of an ACTIVERECORD developer.

For a database instance size of scale factor 1, the query that calculates the outermost nesting level returns nearly 1, 500 tuples, each constiting of a `user_id` and a surrogate value for the associated orders.

The query responsible for the second nesting level delivers 1, 500 `order_ids` and the overall costs that could be saved by choosing the cheapest variants. A surrogate value again enables us to reference the values of the innermost nesting level.

The third query in turn returns 6, 000 tuples with the associated line items the chosen variants and their cheapest alternatives. The results are then materialized, and assembled in the host language in order to derive the nested result shape. The loop-lifted compilation ensures that all values are delivered in the proper logical order. Consequently, we can derive the nested result shape in linear time, depending on the result size.

```
orders = Orders.select { |os| os.checkout_complete = 1 }

orders.group_with(&:user_id).map { |uid, os|
  { user   : uid,
    orders :
      os.map { |o|
         suggestions = o.line_items.map { |li|
             yvar = li.variant
             svar = Variants.select { |v|
                       v.product_id == yvar.product_id
                    }.min_by(&:price)
             { lineitem : li.id,
               your_var : { id : yvar.id, price : yvar.price },
               sugg_var : { id : svar.id, price : svar.price } }
         }

         { order       : o.id,
           suggestions : suggestions,
           saving      : suggestions.map { |v|
                           v.your_var.price - v.sugg_var.price
                         }.sum }
      } }
}
```

Figure A.3: Variant of Query B_5 formulated from the angle of a RUBY purist using SWITCH.

In the SQL variant, shown in Figure A.4, we derive the relevant information in a single query. In order to ensure that only one cheapest variant is suggested for each line item, we use the `ROWNUMBER()` facility (see line 17) along with the predicate in line 28.

For an instance size of scale factor 1, the SQL query sends almost 6,000 tuples to the host language. Observe that grouping is completely left to the host language (consider the red code fragments) to assemble the nested result shape.

```
query = <<-SQL
WITH

PurchasedVariants(id, product_id, price, li_id, order_id, user_id) AS
  (SELECT v.id, v.product_id, v.price, li.id, o.id, o.user_id
     FROM Orders o, Line_Items li, Variants v
    WHERE o.id = li.order_id AND li.variant_id = v.id
      AND o.checkout_complete = 1),

CheapestVariantsID(product_id, price) AS
    (SELECT product_id, MIN(price)
       FROM Variants v
      WHERE v.product_id IN (SELECT product_id FROM PurchasedVariants)
   GROUP BY product_id),

CheapestVariants(rid, id, product_id, price) AS
 (SELECT ROW_NUMBER() OVER (PARTITION BY v.id) AS rid,
         v.id, v.product_id, v.price
    FROM Variants v, CheapestVariantsID cvid
   WHERE v.price = cvid.price AND v.product_id = cvid.product_id),

Suggestions(sid, sproduct_id, sprice, id, product_id,
            price, li_id, order_id, user_id) AS
  (SELECT cv.id, cv.product_id, cv.price, pv.id, pv.product_id,
          pv.price, pv.li_id, pv.order_id, pv.user_id
     FROM CheapestVariants cv, PurchasedVariants pv
    WHERE cv.product_id = pv.product_id
      AND cv.rid = 1),

Savings(order_id, amount) AS
    (SELECT order_id, SUM(s.price - s.sprice) AS amount
       FROM Suggestions s
   GROUP BY order_id)

   SELECT sugg.*, sav.amount
     FROM Suggestions sugg, Savings sav
    WHERE sugg.order_id = sav.order_id
 ORDER BY sugg.user_id;
SQL

ActiveRecord::Base.connection.select(query).
  group_by { |v| v["user_id"] }.map { |uid, o|
    { user: uid,
      orders:
        o.group_by { |v| v["order_id"] }.map { |oid,li|
          suggestions = li.group_by { |v| v["li_id"] }.map { |lid,var|
                          { lineitem : lid,
                            your_var : { id    : var.first["id"].to_i,
                                         price : var.first["price"].to_f },
                            sugg_var : { id    : var.first["sid"].to_i,
                                         price : var.first["sprice"].to_f }
                        }

          { order      : oid,
            suggestions: suggestions,
            savings    : li.first["amount"].to_f }
        } }
  }
```

Figure A.4: Variant of Query B_5 formulated from the angle of database application developer.

Bibliography

[AB87] M. P. Atkinson and P. O. Buneman. “Types and Persistence in Database Programming Languages”. In: *ACM Computing Survey* 19.2 (1987), pp. 105–170.

[AC+11a] P. Alvaro, N. Conway, J. Hellerstein, and W. Marczak. “Consistency Analysis in Bloom: A CALM and Collected Approach”. In: *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*. CIDR 2011. 2011, pp. 249–260.

[AC+11b] J. An, A. Chaudhuri, J. Foster, and M. Hicks. “Dynamic Inference of Static Types for Ruby”. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 2011. 2011, pp. 459–472.

[ACF09] J. An, A. Chaudhuri, and J. Foster. “Static Typing for Ruby on Rails”. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. ASE 2009. 2009, pp. 590–594.

[Act] *ActiveRecord Query Interface*. Jan. 2013. URL: http://api.rubyonrails.org.

[Amb] *Ambition: Database Toolkits for Ruby*. Dec. 2012. URL: http://rubygems.org.

[App04] A. W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 2004. ISBN: 978-0-5216-0765-0.

[BA+09] S. Bellamkonda, R. Ahmed, A. Witkowski, A. Amor, M. Zait, and C. Lin. “Enhanced Subquery Optimizations in Oracle”. In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1366–1377.

[BG+06] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. “MonetDB/XQuery: a Fast XQuery Processor Powered by a Relational Engine”. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD 2006. 2006, pp. 479–490.

[BM02] R. Bayer and E. McCreight. “Software pioneers”. In: New York, NY, USA: Springer-Verlag, 2002. Chap. Organization and Maintenance of Large Ordered Indexes, pp. 245–262.

[BMT07] G. M. Bierman, E. Meijer, and M. T. “Lost in Translation: Formalizing Proposed Extensions to C#”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA 2007. 2007, pp. 479–498.

[BtC+91] V. Breazu-tannen, T. Coquand, C. A. Gunter, and A. Scedrov. “Inheritance As Implicit Coercion”. In: *Information and Computation* 93.1 (1991), pp. 172–221.

[Bus01] 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”. In: *Theoretic Computer Science* 254.1-2 (2001), pp. 363–377.

[But11] D. Butterstein. “Batches: Remote Batch Invocation for Java”. Master Thesis. Tübingen, Germany: Eberhard Karls Universität Tübingen, July 2011.

[BZN05] P. Boncz, M. Zukowski, and N. Nes. “MonetDB/X100: Hyper Pipelining Query Execution”. In: *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*. CIDR 2005. 2005, pp. 225–237.

[CB74] D. D. Chamberlin and R. F. Boyce. “SEQUEL: A Structured English Query Language”. In: *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*. SIGMOD 1974. 1974, 249—264.

[CC77] P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL 1977. 1977, pp. 238–252.

[CI05] W. R. Cook and A. H. Ibrahim. “Integrating Programming Languages & Databases: What is the Problem?” In: *In ODBMS.ORG*. 2005.

[CL+07] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. "LINKS: Web Programming without Tiers". In: *Proceedings of the 5th international Conference on Formal Methods for Components and Objects*. FMCO 2007. 2007, pp. 266–296.

[CM96] L. Cardelli and Abadi M. *A Theory of Objects*. Springer, 1996. ISBN: 978-0-3879-4775-4.

[Cod70] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[Cod82] E. F. Codd. "Relational Database: A Practical Foundation for Productivity". In: *Communications of the ACM* 25.2 (1982), pp. 109–117.

[Coo09] E. Cooper. "The Script-Writer's Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed". In: *Proceedings of the 12th International Symposium on Database Programming Languages*. DBPL 2009. 2009, pp. 36–51.

[Coq] *The Coq Proof Assistant*. Dec. 2012. URL: http://coq.inria.fr.

[Day87] U. Dayal. "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers". In: *Proceedings of the 13th International Conference on Very Large Data Bases*. VLDB 1987. 1987, pp. 197–208.

[Db2a] *DB2 Join Methods*. Dec. 2012. URL: http://pic.dhe.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=\%2Fcom.ibm.db2.luw.admin.perf.doc\%2Fdoc\%2Fc0005314.html.

[Db2b] *db2advis–DB2 Design Advisor Command*. IBM. Dec. 2012. URL: http://pic.dhe.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=\%2Fcom.ibm.db2.luw.admin.cmd.doc\%2Fdoc\%2Fr0002452.html.

[Db2c] ***REORG INDEXES/TABLES*** *Command*. IBM. Dec. 2012. URL: http://pic.dhe.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=\%2Fcom.ibm.db2.luw.admin.cmd.doc\%2Fdoc\%2Fr0001966.html.

[Db2d] ***RUNSTATS*** *Command*. Dec. 2012. URL: http://pic.dhe.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=\%2Fcom.ibm.db2.luw.admin.cmd.doc\%2Fdoc\%2Fr0001980.html.

[DG08] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[Dja] *Django: The Web Framework for Perfectionists with Deadlines*. Jan. 2013. URL: http://www.djangoproject.com.

[Ear] *Operator Details–MSJOIN input argument*. IBM. Jan. 2013. URL: http://pic.dhe.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.gui.doc%2Fdoc%2Fr0021402.html.

[FA+09] M. Furr, J. An, J. Foster, and M. Hicks. "Static Type Inference for Ruby". In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. SAC 2009. 2009, pp. 1859–1866.

[FM08] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008. ISBN: 978-0-5965-1617-8.

[For] *Fortran: Automatic Coding System for the IBM 704*. Oct. 1956. URL: http://www.fortran.com/FortranForTheIBM704.pdf.

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002. ISBN: 978-0-3211-2742-6.

[FR10] M. Fowler and Parson R. *Domain-Specific Languages*. Addison-Wesley, 2010. ISBN: 978-0-3217-1294-3.

[Ful06] H. Fulton. *The Ruby Way*. Addison Wesley, 2006. ISBN: 978-0-6723-2884-8.

[GG+13] G. Giorgidze, T. Grust, A. Ulrich, and J. Weijers. "Algebraic Data Types for Language-Integrated Queries". In: *Proceedings of the 1st International Workshop on Data Driven Functional Programming*. DDFP 2012. 2013.

[GH+94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994. ISBN: 978-0-2016-3361-0.

[GM+07] T. Grust, M. Mayr, J. Rittinger, J. Teubner, and S. Sakr. "A SQL:1999 Code Generator for the Pathfinder XQuery Compiler". In: *Proceedings of the ACM SIGMOD Conference on Management of Data*. SIGMOD 2007. 2007, pp. 1162–1164.

[GM+09] T. Grust, M. Mayr, T. Schreiber, and J. Rittinger. "Ferry: Database-Supported Program Execution". In: *Proceedings of the 28th ACM SIGMOD Int'l Conference on Management of Data*. SIGMOD 2009. 2009, pp. 1063–1066.

[GM12] T. Grust and M. Mayr. "A Deep Embedding of Queries into Ruby". In: *Proceedings of the 28th IEEE International Conference on Data Engineering*. ICDE 2012. 2012, pp. 1257–1260.

[GMR09] T. Grust, M. Mayr, and J. Rittinger. "XQuery Join Graph Isolation". In: *Proceedings of the 25th Int'l Conference on Data Engineering*. ICDE 2009. 2009, pp. 1167–1170.

[GMR10] T. Grust, M. Mayr, and J. Rittinger. “Let SQL Drive the XQuery Workhorse”. In: *Proceedings of the 13th Int’l Conference on Extending Database Technology*. EDBT 2010. 2010, pp. 147–158.

[GR08] T. Grust and J. Rittinger. “Jump Through Hoops to Grok the Loops — Pathfinder’s Purely Relational Account of XQuery-style Iteration Semantics”. In: *Proceedings of the ACM SIGMOD/PODS 5th Int’l Workshop on XQuery Implementation, Experience and Perspectives*. SIGMOD 2008. 2008.

[GRS10] T. Grust, J. Rittinger, and T. Schreiber. “Avalanche-safe LINQ Compilation”. In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 162–172.

[GST04] T. Grust, S. Sakr, and J. Teubner. “XQuery on SQL Hosts”. In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases*. 2004, pp. 252–263.

[Had] *Apache Hadoop: Open-source Software for Reliable, Scalable, Distributed Computing*. Jan. 2013. URL: http://hadoop.apache.org.

[Has] *The Glasgow Haskell Compiler*. Sept. 2012. URL: http://www.haskell.org/ghc/.

[HC97] G. Hamilton and R. Cattell. *JDBC^{TM}: A Java SQL API*. Sun Microsystems Inc. Jan. 1997. URL: http://www.dcs.ed.ac.uk/teaching/cs2/prac6/jdbc-spec-0120.pdf.

[Hud98] P. Hudak. “Modular Domain Specific Languages and Tools”. In: *Proceedings of the 5th International Conference on Software Reuse*. ICSR 1998. 1998, pp. 134–142.

[IB+07] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. “Dryad: Distributed Data-parallel Programs from Sequential Building Blocks”. In: *SIGOPS Operating Systems Review* 41.3 (2007), pp. 59–72.

[IJ+09] A. H. Ibrahim, Y. Jiao, E. Tilevich, and W. R. Cook. “Remote Batch Invocation for Compositional Object Services”. In: *Proceedings of the 23rd European Conference on Object-Oriented Programming*. ECOOP 2009. 2009, pp. 595–617.

[Jav] *Java Stanard Edition*. Jan. 2013. URL: http://www.oracle.com/technetwork/java/javase/overview/index.html.

[Jdb] *Sun Microsystem: JDBC Overview*. Jan. 2013. URL: http://www.oracle.com/technetwork/java/javase/jdbc/index.html.

[Jon03] S. P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Simon Peyton Jones, 2003. ISBN: 978-0-5218-2614-3.

[Kan08] T. Kandler. “Ein kdb+-Code-Generator für den Pathfinder XQuery Compiler”. Master Thesis. München, Germany: Technische Universität München, Jan. 2008.

[KB+07] D. Kulkarni, L. Bolognese, M. Warren, A. Hejlsberg, and K. George. LinQ *to SQL:* .Net *Language-Integrated Query for Relational Data*. Tech. rep. Microsoft Corporation, 2007.

[Kim82] W. Kim. “On Optimizing An SQL-like Nested Query”. In: *ACM Transactions on Database Systems (TODS)* 7.3 (1982), pp. 443–469.

[KS96] G. Keller and M. Simons. “A Calculational Approach to Flattening Nested Data Parallelism in Functional Languages”. In: *Proceedings of the 2nd Asian Computing Science Conference on Concurrency and Parallelism, Programming, Networking, and Security*. ASIA 1996. 1996, pp. 234–243.

[Lin] *LINQ to SQL*. Microsoft Corporation. Aug. 2012. URL: http://msdn.microsoft.com/en-en/library/bb386976.aspx.

[Mai90] D. Maier. “Representing Database Programs as Objects”. In: *Advances in Database Programming Languages*. New York, NY, USA: ACM, 1990, pp. 377–386.

[May07] M. Mayr. “Ein SQL:1999 Generator für Pathfinder”. PhD thesis. München, Germany: Technische Universität München, Apr. 2007.

[May08] M. Mayr. “Pathfinder meets DB2”. In: *Ph.D. Workshop of the 11th Int’l Conference on Extending Database Technology*. EDBT 2008. 2008, pp. 59–64.

[McK65] W. M. McKeeman. “Peephole Optimization”. In: *Communications of the ACM* 8.7 (1965), pp. 443–444.

[Mel02] J. Melton. *Advanced SQL:1999 - Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann, 2002. ISBN: 978-1-5586-0677-7.

[Mey98] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1998. ISBN: 978-0-1362-9155-8.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire”. In: *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. FPCA 1991. 1991, pp. 124–144.

[Ms] *Microsoft Open Database Connectivity*. Microsoft Corporation. Dec. 2012. URL: http://msdn.microsoft.com/en-us/library/ms710252(v=vs.85).

[MS01] J. Melton and A. R. Simon. *SQL:1999 - Understanding Relational Language Components*. Morgan Kaufmann, 2001. ISBN: 978-1-5586-0456-8.

[OR+08] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. “Pig Latin: A Not-so-foreign Language for Data Processing”. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD 2008. 2008, pp. 1099–1110.

[PD+05] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. “Interpreting the Data: Parallel Analysis with Sawzall”. In: *Scientific Programming* 13.4 (2005), pp. 277–298.

[Per] PERL*'s Database Interface*. Jan. 2013. URL: `http://dbi.perl.org`.

[Pie02] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN: 978-0-2621-6209-8.

[Pie91] B. C. Pierce. “Programming with Intersection Types and Bounded Polymorphism”. Ph.D. Thesis. Pittsburgh, PA 15213: Carnegie Mellon University, Dec. 1991.

[Pyt] *Python Programming Language*. Jan. 2013. URL: `http://www.python.org`.

[Rit10] J. Rittinger. “Constructing a Relational Query Optimizer for Non-Relational Languages”. Ph.D. Thesis. Tübingen, Germany: Eberhard Karls Universität Tübingen, Apr. 2010.

[Rub] *Ruby on Rails: Web Development that Doesn't Hurt*. Dec. 2012. URL: `http://rubyonrails.org`.

[SA+10] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. “MapReduce and Parallel DBMSs: Friends or Foes?” In: *Communications of the ACM* 53.1 (2010), pp. 64–71.

[SB+10] T. Schreiber, S. Bonetti, T. Grust, M. Mayr, and J. Rittinger. “Thirteen New Players in the Team: A Ferry-based LINQ to SQL Provider”. In: *Proceedings of the 36th International Conference on Very Large Data Bases*. VLDB 2010. 2010, pp. 1549–1552.

[Sch08] T. Schreiber. “Translation of List Comprehensions for Relational Database Systems”. Master Thesis. München, Germany: Technische Universität München, Mar. 2008.

[Spr] *Spree: The World's Most Flexible E-Commerce Platform*. Jan. 2013. URL: `http://spreecommerce.com`.

[SS86] H. J. Schek and M. H. Scholl. “The Relational Model with Relation-valued Attributes”. In: *Information Systems* 11.2 (1986), pp. 137–147.

[SS90] M. H. Scholl and H. J. Schek. “A Relational Object Model”. In: *Proceedings of the 3rd International Conference on Database Theory.* ICDT 1990. 1990, pp. 89–105.

[SS91] M. H. Scholl and H. J. Schek. “From Relations and Nested Relations”. In: *Proceedings of the 9th British National Conference on Databases.* BNCOD 1991. 1991, pp. 202–225.

[Sym06] D. Syme. “Leveraging .NET Meta-programming Components from F#: Integrated Queries and Interoperable Heterogeneous Execution”. In: *The 2006 ACM SIGPLAN Workshop on ML.* ML 2006. 2006.

[Teu06] J. T. Teubner. “Pathfinder: XQuery Compilation for Relational Database Targets”. Ph.D. Thesis. München, Germany: Technische Universität München, Sept. 2006.

[Tor06] M. Torgersen. “Language Integrated Query: Unified Querying across Data Sources and Programming Languages”. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications.* OOPSLA 2006. 2006, pp. 736–737.

[TP89] P. Trinder and Wadler P. “Improving List Comprehension Database Queries”. In: *Proceedings of the IEEE Region 10 Conference.* TENCON 1989. 1989, pp. 186–192.

[Tpc] *TPC Benchmark H.* Jan. 2013. URL: http://www.tpc.org/tpch/.

[TS+09] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. “Hive: A Warehousing Solution over a Map-reduce Framework”. In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1626–1629.

[Ulr11] A. Ulrich. “A Ferry-Based Query Back-end for the Links Programming Language”. Master Thesis. Tübingen, Germany: Eberhard Karls Universität Tübingen, Mar. 2011.

[VP95] M. Venkatrao and M. Pizzo. “SQL/CLI — A New Binding Style for SQL”. In: *ACM SIGMOD Record* 24.4 (1995), pp. 72–77.

[Wad92] P. Wadler. “Comprehending Monads”. In: *Mathematical Structures in Computer Science.* MSCS 1992. 1992, pp. 61–78.

[WC07] B. Wiedermann and W. R. Cook. “Extracting Queries by Static Analysis of Transparent Persistence”. In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL 2007. 2007, pp. 199–210.

[Wita] *Common Table Expressions.* IBM. Dec. 2012. URL: http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=\%2Fcom.ibm.db2.doc.sqlref\%2Fsscte.htm.

[Witb] *Subquery Factoring.* Oracle Corporation. 2010. URL: http://docs.oracle.com/cd/B28359_01/server.111/b28286/statements_10002.htm.

[Witc] *WITH Queries.* Dec. 2012. URL: http://www.postgresql.org/docs/9.1/static/queries-with.html.

[WS07] G. Wassermann and Z. Su. "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities". In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI 2007. 2007, pp. 32–41.

[XP99] H. Xi and F. Pfenning. "Dependent Types in Practical Programming". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL 1999. 1999, pp. 214–227.

[YI+08] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. "DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation.* OSDI 2008. 2008, pp. 1–14.