



Universität Tübingen | 72074 Tübingen | Germany

**Mathematisch-
Naturwissenschaftliche
Fakultät**

Datenbanksysteme

Dennis Butterstein

d.butterstein@web.de

2011

Batches

Master's Thesis at University of Tübingen

Verfasst und vorgelegt von:

Dennis Butterstein

Betreuer:

Prof. Dr. Torsten Grust

Manuel Mayr

Fassung May 27, 2011

© 2011 Dennis Butterstein

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 2.0 License. To view a copy of this license, visit

<http://creativecommons.org/licenses/by-nc-sa/2.0/de/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Satz: PDF-L^AT_EX 2_ε

Contents

List of Figures	vi
List of Tables	viii
Listings	ix
1 Introduction	1
1.1 Old-Fashioned Way of Integrating Databases	1
1.2 Batches' Way of Integrating Databases	2
1.3 Ferry-Integration	3
2 Batches Basic System	6
2.1 The Batches Language	6
2.2 Operators Implemented by the Batches System	7
2.2.1 Binary and Unary Operations	8
2.2.2 Aggregation Operations	9
2.2.3 Set Operations	10
2.3 Implementation types	11
2.4 Translationrelevant Classes	11
2.4.1 Frontend	12
Jabac	12
Defining a Database Schema	13
2.4.2 Backend	15
3 Target System	16
3.1 Motivation	16
3.2 Pathfinder	17
3.2.1 Relational Data Model	17
Flat Lists	17
Nested Lists	17
Records	18
3.2.2 Implementation Types	18

Contents

- 3.2.3 Relational Algebra 19
 - Serialize Relation 19
 - Literal Table 21
 - Empty Table 21
 - Reference Table 21
 - Attach 21
 - Cross 22
 - Theta Join and Equi Join 22
 - Project 22
 - Select 23
 - Union 23
 - Intersect 23
 - Difference 24
 - Distinct 24
 - Fun_1to1 24
 - Aggregation 25
 - Rownumber 25
 - Rowrank 26
- 4 Translation 27**
 - 4.1 Introduction 27
 - 4.1.1 Notation 27
 - 4.1.2 Handling Inner Tables 28
 - Adhere Inner Tables 28
 - Filter Inner Tables 31
 - 4.2 Normalize Input Tree 34
 - 4.2.1 TableDot 34
 - 4.3 Type Inference 35
 - 4.4 Translate to Table Algebra 40
 - Output 40
 - Record 40
 - Let 41
 - OrderBy 41
 - For 42
 - Count 44
 - Average, Sum, Min, Max 46
 - All 49
 - Exists 52
 - Field access 53

Contents

Project	54
First	54
Box	55
Unbox	57
Distinct	57
Union	58
Intersect	59
Where	59
Constant	61
Binary and Unary Operations	63
Var	64
TableDot	65
4.5 Translate to SQL	66
4.6 Providing results	68
5 Conclusion	71
Bibliography	72

List of Figures

1.1	Old-fashioned way of using databases	2
1.2	BATCHES' way of using databases	3
1.3	BATCHES including Ferry	4
2.1	Grammar of BATCHES	7
2.2	Binary and unary operations	8
2.3	Aggregation operations	9
2.4	Set operations	10
2.5	Implementation types	11
2.6	Structure of schema definitions	13
2.7	Northwind interfaces structure	14
2.8	Definition of the <i>Forest</i> data structure	15
3.1	Flat list [10, 20, 30]	17
3.2	Nested list [[], [10, 20, 30], [40, 50]]	18
3.3	Record [(a,10), (b,20), (c,30)]	18
3.4	Implementation types of BATCHES vs. relational implementation types	19
3.5	Example: \oplus operator	21
3.6	Example: $@$ operator	21
3.7	Example: \times operator	22
3.8	Example: \otimes operator	22
3.9	Example: π operator	23
3.10	Example: σ operator	23
3.11	Example: \cup operator	23
3.12	Example: \cap operator	24
3.13	Example: \setminus operator	24
3.14	Example: δ operator	24
3.15	Example: \odot operator	25
3.16	Example: $\#$ operator	25
3.17	Example: $\#$ operator	26
3.18	Example: <i>rnk</i> operator	26
4.1	Example: Adherence of outer and inner tables	30

List of Figures

4.2	Example: Adherence of inner tables	31
4.3	Example: Filter tables	33
4.4	Example: OrderBy	43
4.5	Example: count(λ)	47
4.6	Example: Restoring empty iterations	48
4.7	Example: Creation of q'	50
4.8	Example: sum	51
4.9	Example: all	53
4.10	Example: First	56
4.11	Example: Box	57
4.12	Example: Unbox	58
4.13	Example: Union	60
4.14	Example: Where	62
4.15	Example: Translation of a constant	62
4.16	Example: Translation of binary operations	63
4.17	Example: \boxtimes operator	65
4.18	Structure of query plan bundles	66
4.19	Structure of PATHFINDERS' result	67
4.20	Definition of the <i>Forest</i> data structure	68

List of Tables

3.1	Operators and semantics of the table algebra	20
4.1	Representation of missing comparison operations	64

Listings

2.1	<i>Jabac</i> input	12
2.2	<i>Jabac</i> output	12
2.3	Class Product.java	14
3.1	Mapping of the implementation types	19

1 Introduction

The query-language BATCHES, developed at the University of Texas, provides developers with a simple yet powerful way of processing database-located data without writing a single line of SQL¹, remote procedure calls and webservice. So developers do not have to switch paradigms (in mind) from an object oriented language, such as Java to the relational paradigm of SQL. By using native Java syntax, developers do not even have to get used to any new kind of commands except specific ones to simplify handling (such as print) resulting in a seamless way of integrating database operations. Because of its architecture the BATCHES-System brings some improvements. By executing database queries in BATCHES it is able to reduce the count of round-trips between database server and client as well as traffic produced by multiple interferences. Knowing that using Batches seems to be an attractive way of using databases in applications and benefit from its advantages. A major drawback certainly is that BATCHES does not support lots of possible operations yet. To solve that problem in the following we try to integrate FERRY compilation techniques with BATCHES in order to extend the programmers combinator toolkit as well as the performance.

1.1 Old-Fashioned Way of Integrating Databases

The most basic way to integrate databases with programming languages is to use drivers (such as JDBC) to execute queries on a database and receive the results. Without using persistence frameworks or similar, the programmer would have to manually write the SQL queries needed for every use case, execute them, get the results from the database and finally process the received data in the heap of the programming language in order to get the result required. Furthermore the data delivered in the resultset would be "raw" when not using any kind of ORM.

Persistence frameworks, such as hibernate or castor, are able to simplify database handling especially for "standard" queries such as loading all rows of a table, but for special purposes programmers still have to write queries explicitly for a given

¹Structured Query Language

1 Introduction

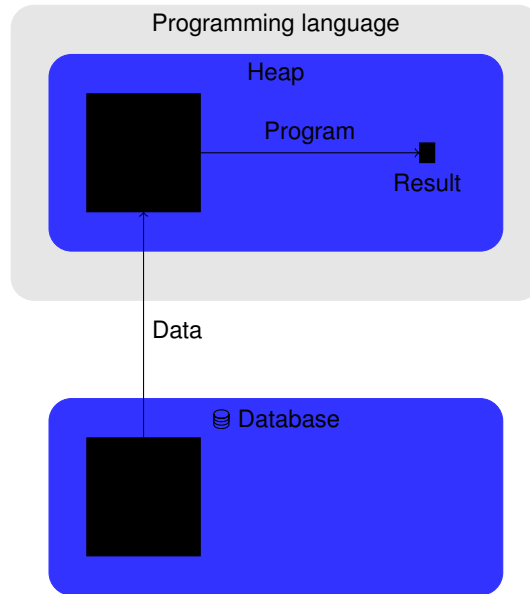


Figure 1.1: Old-fashioned way of using databases

case. These frameworks are able to ease the need to handle "raw" data from resultsets by using ORM-techniques as well but classes have to be defined for every table that map rows of that table to an object of the programming language. Considering above facts, the usage of databases is mainly restricted on loading data processed in the heap of the programming language without using the whole processing power modern databases offer.

1.2 Batches' Way of Integrating Databases

The BATCHES' way of integrating database processing in programming languages is somewhat different. The programmer does not need to write SQL himself - semantically extended expressions are used - for example a for-loop - to interact with the database. These extended expressions are compiled to the BATCHES language out of which SQL queries are generated being executed on the database. Those queries are generated with the intention to process as much work as possible directly on the database system and deliver the concrete result of the written batch. The returned results are written back to a data structure (called *forest*) to make them accessible for the underlying programming language.

So from the programmers view it must not be dealt with "raw" results and not a single line of SQL must be written. As a consequence programmers can focus on writing well-structured programs, need not worry about writing efficient SQL queries

1 Introduction

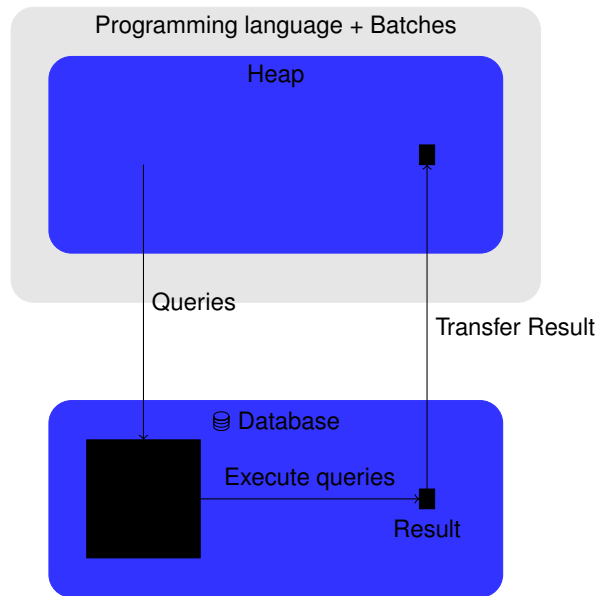


Figure 1.2: BATCHES' way of using databases

and do not transfer superfluous data into the programming language's heap.

1.3 Ferry-Integration

One may ask what FERRY has to do with BATCHES. Actually their basic idea is very similar - take the work to do and delegate it to the underlying database system. In fact FERRY tries to push this idea even one step further - delegate as much work as possible to the underlying database system and execute it in an efficient way. Since database systems are well understood and highly optimized for processing huge amounts of data it is a reasonable decision to integrate them in program execution.

When looking at figure 1.3 the question arises why should we integrate another step in the process of creating queries? As FERRY replaces the generation of queries provided by BATCHES there should be several benefits. First of all the FERRY language does support a richer set of operations than BATCHES right now. So a major benefit should be to achieve a larger combinator toolkit. Another point why integrating FERRY is reasonable is that the generated queries will run on different database systems, such as *IBM DB2*, *MS SQL Server* or *PostgreSQL*, out of the box. So using FERRY should add flexibility concerning the database used. Additionally the queries generated by FERRY are highly optimized, so they should perform well when executed on a database. As BATCHES operates on sets it does not care about the order of the results. In contrast FERRY operates on ordered

1 Introduction

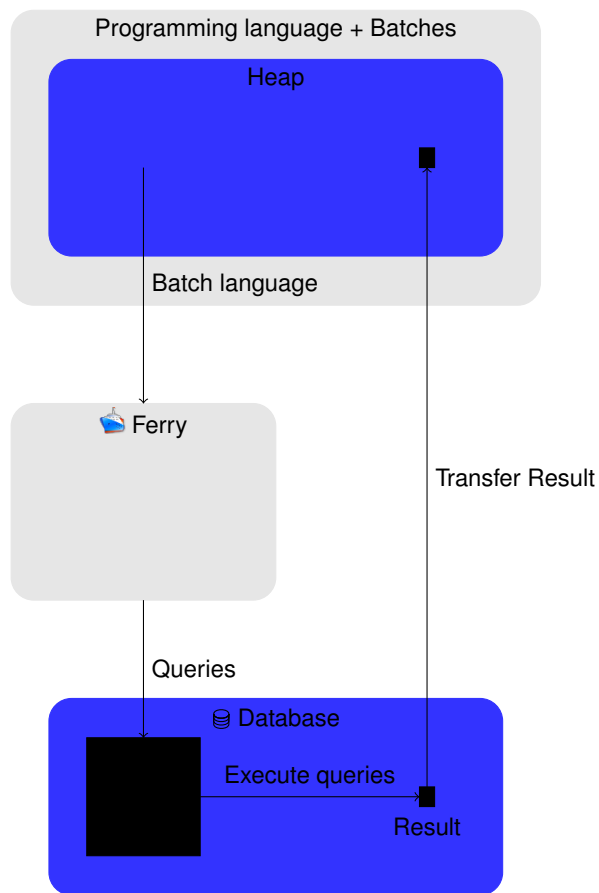
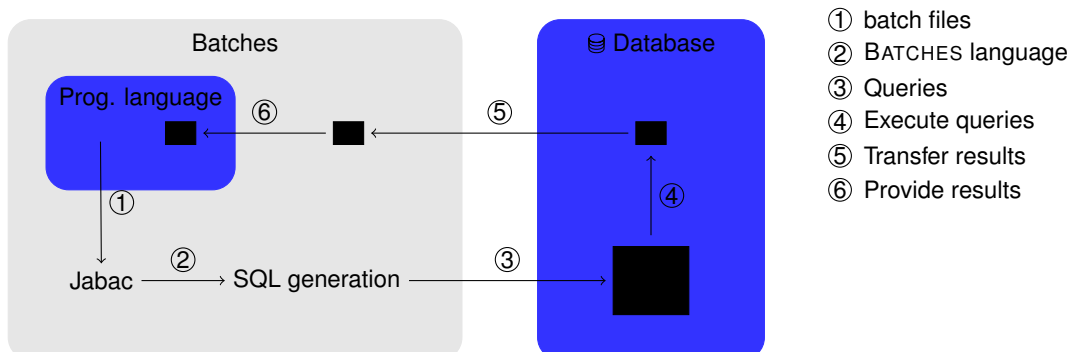


Figure 1.3: BATCHES including Ferry

1 Introduction

lists, so using `FERRY` adds the possibility to rely on the order of the results.



2 Batches Basic System

The BATCHES basic system we integrated FERRY into, provided us with some out of the box features we use as interfaces to insert FERRY. First of all BATCHES contains a compiler *Jaba* that supports the compilation of a subset of the *Java* language that is essential for compiling written batches. Actually *Jaba* does not change the native *Java* syntax, *Jaba* rather extends the semantics of *Java*'s for-loops to be able to invoke batch services. *Jabas* output will be a program formulated in the BATCHES language serving as the base language to generate SQL queries from. Furthermore the results of the generated queries are written back to a datastructure called *Forest* that comes with BATCHES. Knowing that the frontend interface for FERRY to interact with, will be *Jabac* with its output language BATCHES and the backend interface is provided by the *Forest* data structure.

2.1 The Batches Language

As input for query construction the BATCHES language is used. Its grammar is defined in Figure 2.1 as stated in [3]. Language constructs highlighted in blue have been implemented in the FERRY integration approach. The language is of compositional character - so it can be composed in every imaginable way. Compositionality is an important feature for the input language since the target language is compositional as well. So in order not to restrict the hitherto expressiveness of the target language, the input language has to be compositional, too.

2 Batches Basic System

The core construct of the language, dealing with sets is given by the *for* expression. It is able to iterate over a set and apply a given expression to each element contained. Besides that, the *for* expression can be used to apply operations over the whole set, such as summing up the elements of all values or calculate the average over the set. Note that we are not able to predict the order of any result as we are operating on sets that do not preserve any order.

The expression $e.m(\bar{e})$ enables us to call a method on an expression. The peculiar about it is that the method can be an arbitrary string, so theoretically it is possible to call any imaginable method on an expression. In practice the method has to be implemented in the translation, otherwise it cannot be properly invoked.

e	$::=$	x	variable
		$ $	
		c	constant
		$ $	
		$e.f$	access a field
		$ $	
		$e.m(\bar{e})$	invoke method
		$ $	
		$\odot(\bar{e})$	primitive operation
		$ $	
		$v \odot = \bar{e}$	variable assignment
		$ $	
		$e.f \odot = f$	field assignment
		$ $	
		let $\bar{x} \equiv \bar{e} e$	variable binding
		$ $	
		for $\odot(x \in e) e$	for-loop
		$ $	
		try $e x e$	exception handling
		$ $	
		$\lambda x.e$	first-class function
		$ $	
		l	input parameter with name l
		$ $	
		e^l	output result with name l
\odot	$::=$	$+ - * \backslash mod = ! != < >$	
		$ $	
		$\leq \geq \vee \wedge \dots$	
x, m, f	$::=$	name	
c	$::=$	Null Boolean String Integer Float	possible types for constants
		$ $	
		Decimal Date Time Datetime RawData	

Figure 2.1: Grammar of BATCHES

2.2 Operators Implemented by the Batches System

The BATCHES system provides a set of operators implemented in the compiler as well as in the backend system. This set of operators - introduced in the following - were consulted to be implemented in the FERRY integration approach.

2.2.1 Binary and Unary Operations

Binary operations serve as functions taking two operands and combining them in the way dictated by the specific operator. When using them we have to be aware of the datatypes the operands are instances of. If the operand-types are not equal we could have to cast them in order to combine them with the binary operator given, if possible. Another point we have to consider is that some of the operators are commutative whilst others are not. Unary operations only take one operand.

Operator	Description
Binary	
Arithmetic	
+	Addition of two operands
-	Subtraction of two operands
×	Multiplication of two operands
÷	Division of two operands
mod	Modulo of two operands
Comparison	
=	Tests equality of two operands
≠	Tests inequality of two operands
<	Tests if first operand is lower than the second
>	Tests if first operand is greater than the second
≤	Tests if first operand is lower or equal to the second
≥	Tests if first operand is greater or equal to the second
Logical	
∧	Logical <i>and</i> between the two operands
∨	Logical <i>or</i> between the two operands
Unary	
Logical	
¬	Negates its boolean input

Figure 2.2: Binary and unary operations

The binary operators as supplied by BATCHES are listed in Figure 2.2. As you may have expected all of the listed operators are primitive operators. In theory the BATCHES language is able to handle n-ary operations directly, but it seems as if n-ary operations are expressed as a sequence of binary expressions - so we list the operations as unary and binary ones.

2.2.2 Aggregation Operations

In general aggregation operations combine a batch of data to receive new results such as average values, sums or similar achievements. Most aggregation operations in BATCHES are implemented as higher order functions, so they take a function as an argument. This argument function has a specified return type as well as a specified input type - so the function prescribes the types the aggregation operation is able to handle. By using that technique it is, for example, possible not only to sum up all values of a specific column but also to multiply the value of each row with a constant before summing them up. So this approach turns out to be a very flexible one. The operators implemented are shown in Figure 2.3.

Operator	Description
count()	Number of input rows
count($\lambda x.e :: \text{Boolean}$)	Number of items satisfying given function
average($\lambda x.e :: \text{Double}$)	Average of the set after applying the given function to each item
sum($\lambda x.e :: \text{Long}$)	Sum of the set after applying the given function to each item (return type Long)
dsum($\lambda x.e :: \text{Double}$)	Sum of the set after applying the given function to each item (return type Double)
min($\lambda x.e :: \text{Long}$)	Minimum of the set after applying the given function to each item (return type Long)
dmin($\lambda x.e :: \text{Double}$)	Minimum of the set after applying the given function to each item (return type Double)
max($\lambda x.e :: \text{Long}$)	Maximum of the set after applying the given function to each item (return type Long)
dmax($\lambda x.e :: \text{Double}$)	Maximum of the set after applying the given function to each item (return type Double)
exists()	Checks the underlying set for emptiness and returns a boolean value
exists($\lambda x.e :: \text{Boolean}$)	Applies function to the items in order to decide whether at least one of them satisfies the given function

Figure 2.3: Aggregation operations

Count can be invoked with as well as without function passed as an argument. Here the function is used to specify what arguments we have to count. If there is

2 Batches Basic System

no function given we simply count every item in the set. The semantics of exists is very similar. Invoked without a function, exists checks if the underlying set is empty only. Given a function it has to process the items and search for one of them matching the condition of the function. The operations sum, min and max are doubly configured to cover the operations on Double as well as Long types. The only difference is the type they return and operate on.

2.2.3 Set Operations

Set operations operate on a set and their result will be a new set with the items the operation was applied on. In Figure 2.4 the implemented set operations are presented.

Operator	Description
distinct()	Removes duplicates from underlying set
first(n)	Prunes input set to the first n elements
where($\lambda x.e :: \text{Boolean}$)	Checks if items satisfy given function and removes those that do not
orderBy($\lambda x.e :: P, \text{direction}$)	Orders the underlying set by the returned values of the function (type P) in the given direction (ascending, descending)
groupBy($\lambda x.e :: P$)	Groups the items of the set by the attribute of type P
project($\lambda x.e :: P$)	Restricts the output of a given set to an attribute of type P the items hold
union(Set)	Merges containments of underlying and given set
intersect(Set)	Prunes the output to the items both sets contain

Figure 2.4: Set operations

Distinct seems to be a superfluous operator on sets, but it actually is needed to prune duplicates as database results may contain duplicates. The first operator returns the first n items of a set - since we cannot predict the order the database returns the results (except orderBy is executed explicitly) we don't know what n items will be the result of the first operation. At first glance the orderBy operation makes no sense as we operate on sets - on closer inspection we need the orderBy operator as we are able to execute it on the database - so results we receive are ordered by the database. The returned results we use preserve that order - so we are able to use orders but only when explicitly executed on the database.

2 Batches Basic System

GroupBy groups the items in the underlying set by the returned values of the given function. So all items in the set returning the same value will be added to the same group, using the return values of the function as keys. The result will be a list of groups where all items are sorted into by the key. The union and intersect operators operate on a set and take a set as argument. Note that duplicate values are not allowed to be contained in a set.

2.3 Implementation types

The implementation types represent the datastructures BATCHES implicitly operates on. In fact BATCHES operates on datastructures that do not preserve any order throughout the representation such as sets and multisets. So we neither are able to track the order of any values during the translation process nor can we receive duplicate values because sets, per definition, are duplicate free. Since orders as well as the explicit handling of duplicates are vital aspects of the FERRY translation approach, we define the datastructures used as lists and records. Figure 2.5 illustrates the used implementation types.

$$\begin{aligned} Record & ::= (\overline{Item}) \\ Item & ::= List \mid Atom \\ List & ::= [Record] \\ Atom & ::= String \mid Integer \mid Float \mid \dots \end{aligned}$$

Figure 2.5: Implementation types

Records consist of one or many *Items*. An *Item* can be represented by a *List* or an *Atom*. *Atoms* are literal values of type string, integer, float or any other primitive type. *Records* can not be nested, but may contain *Lists*. *Lists* are built of a *Record*. So we are able to nest *Lists* in *Records* to arbitrary depth. As *Lists* preserve orders, we are able to use positional accesses and can rely on a specific order during the FERRY translation approach.

2.4 Translationrelevant Classes

To be able to understand where FERRY compilation techniques are integrated in the BATCHES system one has to be familiar with the interfaces of the frontend and

2 Batches Basic System

the backend of BATCHES we use. Below the classes needed to understand the subsequent chapters are introduced.

2.4.1 Frontend

Under the term *frontend* I summarize all parts of the BATCHES framework that are needed to get BATCHES properly working.

Jabac

The first step in the translation process to generate queries is performed by the BATCHES framework in form of the *Jabac* compiler. It translates the Java classes enriched with special semantic commands to abstract syntax trees (subsequently AST) describing the grammar of the BATCHES language as stated in Figure 2.1. In the following these trees serve as the input for query generation. To construct these *ASTs*, *Jabac* takes input files defining batch statements, replaces specific parts of the code and finally compiles them to *Java* class files.

Such an input file could look as Listing 2.1 shows.

```
1 public void add() {
2     BatchFactory f = connection.getFactory();
3     batch.util.Forest s = new batch.util.Forest();
4     batch.util.Forest r = connection.execute(
5         f.Loop(batch.Op.SEQ, "p", f.Prop(f.Root(), "Products"),
6             f.Out("g121", f.Prim(batch.Op.ADD, f.Prop(f.Var("p"),
7                 "UnitsOnOrder"), f.Data(10))))), s);
8     for (batch.util.Forest p : r.getIteration("p"))
9         print(p.getInteger("g121"));
10 }
```

Listing 2.1: *Jabac* input

Listing 2.2: *Jabac* output

Semantically the defined input batch selects column *UnitsOnOrder* from Table *Products*, adds 10 to every value and prints the result to stout.

After the file has been processed by *Jabac* it would look as shown in Listing 2.2.

The result of the compilation by *Jabac* is a class enriched with code constructing the *AST* we use as input for our compilation. The `batch` keyword, followed by the iteration over an implemented root service interface (here `connection`) serves as a hook indicating that the following `body` of the statement is a batch and has to be compiled by *Jabac*. The body can contain both, remote and local statements. At compile time *Jabac* replaces the body with a call to the `execute` method of the

2 Batches Basic System

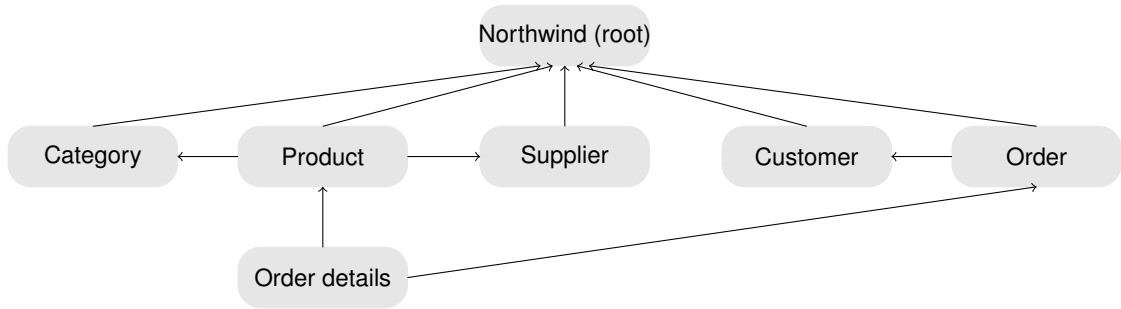


Figure 2.7: Northwind interfaces structure

is returned, a string that denotes the name of the function and a parameter which has an arbitrary name and a prescribed *Type* the parameter has to be of.

In our concrete *Java* context, we use annotated abstract classes defining the columns, keys, foreign keys, sets and functions that are available for the specific tables. In the *Northwind* context these classes are *Northwind.java*, *Product.java*, *Order.java*, *Order_Details.java*, *Supplier.java*, *Category.java* and *Customer.java*. This set of defined interfaces and their relations are shown in Figure 2.6.

The *Northwind* class defines the root interface and as such references every other interface defined. As illustration Listing 2.3 shows an excerpt of the interface class *Product* of the concrete *Java* implementation.

```
1 @Table(name="Products")
2 public abstract class Product {
3
4     @Id
5     public int ProductID;
6
7     public String ProductName;
8
9     ...
10
11     @Inverse("Product")
12     public Set<Order_Details> OrderDetails;
13
14     static Fun<Product, Boolean> isOutOfStock = new Fun<Product, Boolean>() {
15         public Boolean apply(Product p) {
16             return p.UnitsInStock == 0;
17         }
18     };
19
20     ...
21
```

22 }

Listing 2.3: Class Product.java

The lines 5 and 7 demonstrate the definition of fields with primitive datatypes whereas lines 14 – 18 show the definition of a first-class function.

2.4.2 Backend

The *Forest* is the universal datastructure being filled with the results of the queries executed - as such it represents the backend interface of the translation process providing the processed results to the programming language used.

$F \in Forest$	$::= MV MAF F \overline{M}$
$MF \in MultiForest$	$::= LF F$
$MV \in Values Map$	$::= [I \rightarrow T O]$
$MAF \in MultiForest Map$	$::= [I \rightarrow MF]$
$LF \in Forest List$	$::= [\overline{F}]$
$M \in Method$	$::= T m(\overline{T p});$
$T \in Type$	$::= P I$
$P \in Primitive$	$::= Void Number Boolean String$ $ A Duration RawData$
$A \in Date$	$::= Date DateTime$
$N \in Number$	$::= Integer Float Decimal$
$I, m, p \in Name$	
$O \in Object$	

Figure 2.8: Definition of the *Forest* data structure

A *Forest's* abstract structure is introduced in Figure 2.8. The main elements a *Forest* consists of are a map of values binding names to corresponding objects, a map binding names to corresponding *MultiForests*, a *Forest* which is considered the parent *Forest* and a set of methods, if needed. The values are of specified type which can be a primitive or a name referencing any defined class. A *MultiForest* consists of a list of *Forests* and a single *Forest* representing the parent. So the map of *MultiForests* serves as a way of representing nested results. As a resultset does not only consist of one resulting row, a *MultiForest* holds a list of *Forests*. All of those *Forests* represent a single row of the resultset - so we are able to iterate over the resulting rows.

3 Target System

This chapter introduces the intermediate language being used in the compilation process between the BATCHES language and the query language to be generated. The intermediate language is a dialect of the relational algebra defining similar operators while semantics sometimes differ and the dialect extends the operator set.

3.1 Motivation

Since integrating new intermediate languages in a compilation process always causes additional overhead, the language integrated has to add significant features and provide major benefits to justify its use. In our case various reasons have to be considered:

- *Database proximity*

Due to the database proximity of the intermediate language, the mapping of the used dialect to the concrete query language is straight forward since conventional query languages already implement the operators used. Because of that the translation from intermediate language to concrete query language is simplified.

- *Exchangeable query language*

The intermediate language enables us to use code generators for different target languages. Thus, we are able to support a wide range of query languages (such as SQL, MIL, etc.) without having to change the intermediate language or any adaptations at the existing implementation. Therefore, a major gain is portability with respect to different database backends.

- *Order*

As the algebra dialect defines operators rownumber (Subsubsection 3.2.3) and rowrank ((Subsubsection 3.2.3) we are able to define and preserve order on relations. Thereby we switch from set to list semantics. Orders are defined in a transient manner - so they are not added to table data but defined and

3 Target System

preserved in memory by using special columns. Consequently, we are able to rely on the order of the results.

According to these attainments, the use of an additional intermediate language seems warrantable.

3.2 Pathfinder

To optimize and translate the generated intermediate language to a proper query language, we use the PATHFINDER compiler, originally developed for the translation of *XQuery*. PATHFINDER provides strategies for algebraic optimizing as well as code generators for several query languages and as consequence allows us to change the underlying database easily.

3.2.1 Relational Data Model

As described in Section 2.3 the main datatypes we operate on are records and lists. Because of that we need a way to represent records as well as lists in our intermediate language. I will introduce the representation subsequently.

Flat Lists

The general data structure we operate on are lists. Lists are represented as tables, each row representing a single value of the list. As lists implicitly contain positional information we preserve that information in the relational data model by using the column *pos* that encodes the position of the item in the list.

pos	item1
1	10
2	20
3	30

Figure 3.1: Flat list [10, 20, 30]

Nested Lists

So far we assumed, that the values represented in lists are atomic - indeed we are able to nest lists. Knowing that we need an appropriate relational representation of nested lists with arbitrary depth.

3 Target System

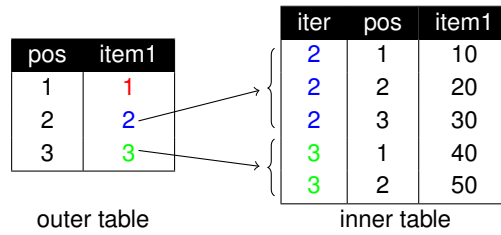


Figure 3.2: Nested list $[[], [10, 20, 30], [40, 50]]$

Inner lists are incorporated by using an outer and an inner table. The outer table encodes the items of the list outer list by introducing a surrogate key for each item and storing their position. Thus, the inner table encodes the inner lists with their values and the position. The *iter* column serves as foreign key to map the items of the inner lists to their position in the outer list. Accessing values of inner lists can be realized by joining the outer and the inner table ($\bowtie_{item1=iter}$). Doing that, empty lists are kept empty, by not providing any inner values in the inner table.

Records

Records of length n are encoded by tables containing columns $item1, \dots, itemn$ that contain the values of the records. Again, the positional information respective the record's location is preserved and reflected in the *pos* column. In contrast to lists, records cannot be nested right now.

pos	item1	item2
1	a	10
2	b	20
3	c	30

Figure 3.3: Record $[(a,10), (b,20), (c,30)]$

3.2.2 Implementation Types

In order to be able to translate the implementation types introduced by BATCHES to the relational world, we have to map them to constructs in relational algebra. In fact, in Section 3.2.1 we already introduced a mapping for records, lists and nested lists - we mapped records to table rows and lists to tables. So the derived implementation type of a record as well as an *Atom* is implementation type *Row*, the derived implementation type for flat lists is *Table*. In contrast, nested lists or records containing lists combine both implementation types, hence they cannot be

3 Target System

assigned to either implementation type *Row* or implementation type *Table* - so they are of implementation type *Row + Table*. The mapping of lists, records and atoms is displayed in Listing 3.1.

```

Record ::= (Item)
Item   ::= List | Atom           t ::= Row   A row of a table
List   ::= [Record]             | Table  {Row}
Atom   ::= String | Integer | Float | ...

```

Figure 3.4: Implementation types of BATCHES vs. relational implementation types

(Atom, ..., Atom) Atom	→	Row
[Record]	→	Table
(..., List, ...) [List]	→	Row + Table

Listing 3.1: Mapping of the implementation types

During the translation process, combinations of the BATCHES language may appear, that implicitly create lists and records that are of implementation type *Row + Table* - so they contain nested lists of implementation type *Table*. As we have to keep the relational data model consistent relating nested lists and records containing lists, we have to handle those cases explicitly. We add a **box** expression to the *AST*, marking that we have to convert the current representation to a inner and outer table (as shown in Section 3.2.1). Another situation that may occur, is that a combination of the BATCHES language is generated, where we need to access the concrete values of an actually nested structure of type *Row + Table*. Again, we need to explicitly handle these situations. Introducing **unbox** calls enables us to mark that we have to merge the outer and inner tables.

3.2.3 Relational Algebra

In the following the target language is introduced by presenting its operations.

Serialize Relation

The serialize operator (\wp) is the root node of an algebra plan. It contains information about the result columns to be aware of their specific function.

3 Target System

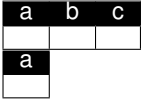
Operator	Semantics
\wp	Serialize relation
	Transient table (literal table)
\emptyset	Empty table
$\boxplus R(c_1, \dots, c_n)$	Persistent table (reference table) with columns c_1 to c_n
$\textcircled{a}_{c:v}$	Attaches column c with value v
\times	Cartesian product
$\bowtie_p, \bowtie_{p_1=p_2}$	Theta join and its special case equi join
$\Pi_{a_1:b_1, \dots, a_n:b_n}$	Projection
σ_p	Select
\cup	Union without duplicate elimination
\cap	Intersection
\setminus	Difference
δ	Distinct
$\odot_{r:(o_1, \dots, o_n)}$	Apply operation
$agg_r(b)/c$	Calculate aggregation
$\#_{r:(o_1:dir, \dots, o_n:dir)/c}$	Calculate rownumber values with respect to grouping c
$rnk_{r:(o_1:dir, \dots, o_n:dir)}$	Calculate rowrank values
$\odot \in \{+, -, \times, \div, \text{mod}, =, \neq, <, >, \leq, \geq, \vee, \wedge\}$ $agg \in \{\text{avg}, \text{max}, \text{min}, \text{sum}, \text{count}, \text{all}, \text{prod}, \text{distinct}\}$ $dir \in \{\text{asc}, \text{dsc}\}$	

Table 3.1: Operators and semantics of the table algebra

3 Target System

Literal Table

A literal table represents a table being transient and as such not persisted in the database. Its content is generated on the fly by the translation.

Empty Table

Empty tables can be thought of as a special case of literal tables. In general they are empty literal tables.

Reference Table

Reference tables represent references to tables persistent in the database.

Products			
id	name	quantity	price
1	Chai	10 boxes × 20 bags	18
2	Chang	24 - 12 oz bottles	19
3	Aniseed Syrup	12 - 550 ml bottles	10

(Persistent table)

\textcircled{P} Products(id,name,quantity,price)	=	<table border="1"> <thead> <tr> <th>id</th> <th>name</th> <th>quantity</th> <th>price</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Chai</td> <td>10 boxes × 20 bags</td> <td>18</td> </tr> <tr> <td>2</td> <td>Chang</td> <td>24 - 12 oz bottles</td> <td>19</td> </tr> <tr> <td>3</td> <td>Aniseed Syrup</td> <td>12 - 550 ml bottles</td> <td>10</td> </tr> </tbody> </table>	id	name	quantity	price	1	Chai	10 boxes × 20 bags	18	2	Chang	24 - 12 oz bottles	19	3	Aniseed Syrup	12 - 550 ml bottles	10
id	name	quantity	price															
1	Chai	10 boxes × 20 bags	18															
2	Chang	24 - 12 oz bottles	19															
3	Aniseed Syrup	12 - 550 ml bottles	10															

Figure 3.5: Example: \textcircled{P} operator

Attach

The attach operator creates a new column for the input relation and assigns every row the same given value. When using attach, the target relation must not have a column with the same name as the column to be attached.

\textcircled{a} <table border="1"> <thead> <tr> <th>a</th> </tr> </thead> <tbody> <tr> <td>1</td> </tr> <tr> <td>2</td> </tr> <tr> <td>3</td> </tr> </tbody> </table>	a	1	2	3	=	<table border="1"> <thead> <tr> <th>a</th> <th>b</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>true</td> </tr> <tr> <td>2</td> <td>true</td> </tr> <tr> <td>3</td> <td>true</td> </tr> </tbody> </table>	a	b	1	true	2	true	3	true
a														
1														
2														
3														
a	b													
1	true													
2	true													
3	true													

Figure 3.6: Example: \textcircled{a} operator

Cross

The cross operator creates the cartesian product of its input relations. That is it combines every row of relation A with each row of relation B . Given a cardinality of a for relation A and b for relation B the cardinality of the resulting relation will be $a \times b$. The amount of columns will be the amount of columns of A plus amount of columns of B , but only if their column names are distinct.

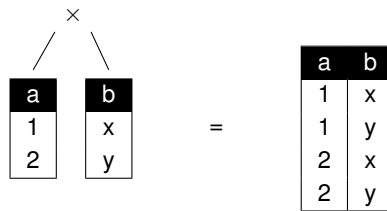


Figure 3.7: Example: \times operator

Theta Join and Equi Join

A theta join (\bowtie_p) combines its input relations with respect to its join predicate p . So all rows matching the predicate are represented in the resulting relation. The predicate p defines one or more comparisons ($=, \neq, <, >, \leq, \geq$) between two columns. The equi join is a special case of the theta join - the predicate p contains a single condition of equality between two columns.

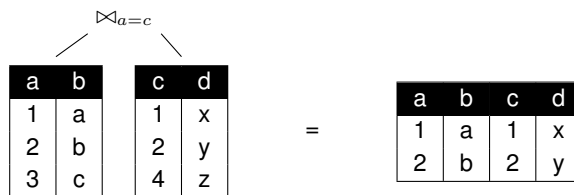


Figure 3.8: Example: \bowtie operator

Project

The project operator removes every column from the input relation not appearing in its column list. The ones appearing may be renamed or the old column name can be kept. Additionally, there is no implicit duplicate elimination as in the classical relational algebra.

3 Target System

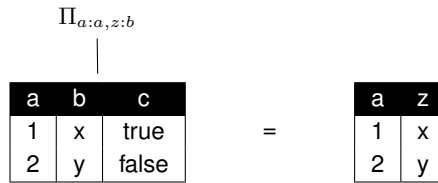


Figure 3.9: Example: π operator

Select

The select operator removes every row from its input relation whose value of the given predicate column p is false. This column has to be of boolean type. There is no implicit duplicate elimination performed.

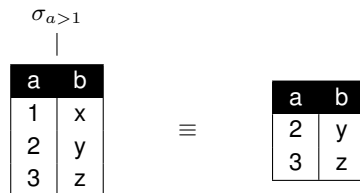


Figure 3.10: Example: σ operator

Union

The Union operator takes two relations as input and merges their rows into a new resulting relation. The column names as well as the types of the columns of the input relations have to be equal. Duplicates in the resulting relation are not removed implicitly.

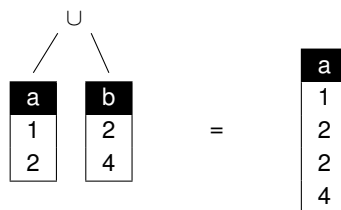


Figure 3.11: Example: \cup operator

Intersect

The intersection operator calculates all rows included in the left hand relation and in the right hand relation. The column names as well as the types of the input

3 Target System

relations have to be equal.

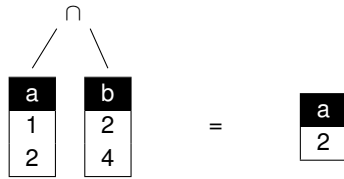


Figure 3.12: Example: \cap operator

Difference

The difference operator calculates the remaining rows after subtraction of the rows of the right hand relation from the rows of the left hand relation. The column names as well as the types of the input relations have to be equal. Additionally, the left relation must not contain duplicates.

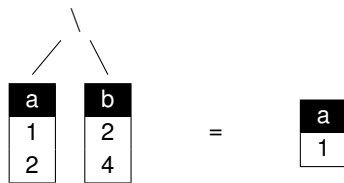


Figure 3.13: Example: \setminus operator

Distinct

The distinct operator removes duplicate rows from the input relation.

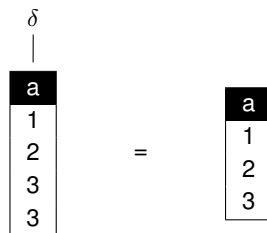


Figure 3.14: Example: δ operator

Fun_1to1

$\text{Fun_1to1} (\odot_{r:(o_1, \dots, o_n)})$ is a generic n-ary operator that, depending on the chosen function, operates on a specific number of inputs. The result of every invocation is

3 Target System

written in a new column r that is attached to the input relation. Possible functions are:

$$\odot \in \{+, -, \times, \div, \text{mod}, =, \neq, <, >, \leq, \geq, \vee, \wedge\}$$

$$\begin{array}{c}
 \dagger_{r:(a,b)} \\
 | \\
 \begin{array}{|c|c|}
 \hline
 \mathbf{a} & \mathbf{b} \\
 \hline
 1 & 10 \\
 2 & 10 \\
 3 & 10 \\
 4 & 10 \\
 \hline
 \end{array}
 \end{array}
 =
 \begin{array}{|c|c|c|}
 \hline
 \mathbf{a} & \mathbf{b} & \mathbf{r} \\
 \hline
 1 & 10 & 11 \\
 2 & 10 & 12 \\
 3 & 10 & 13 \\
 4 & 10 & 14 \\
 \hline
 \end{array}$$

Figure 3.15: Example: \odot operator

Aggregation

The aggregation operation ($agg_{r:(b)/c}$) aggregates the values of column b , grouped by column c . The results are written in the column r that is attached to the input relation. The agg operation can be one of:

$$agg \in \{\text{avg}, \text{max}, \text{min}, \text{sum}, \text{count}, \text{all}, \text{prod}, \text{distinct}\}$$

$$\begin{array}{c}
 agg_{r:(a)/b} \\
 | \\
 \begin{array}{|c|c|}
 \hline
 \mathbf{a} & \mathbf{b} \\
 \hline
 1 & a \\
 2 & b \\
 3 & a \\
 4 & b \\
 \hline
 \end{array}
 \end{array}
 =
 \begin{array}{|c|c|}
 \hline
 \mathbf{b} & \mathbf{r} \\
 \hline
 a & 4 \\
 b & 6 \\
 \hline
 \end{array}$$

Figure 3.16: Example: $\#$ operator

Rownumber

The rownumber operator ($\#_{r:(o_1:dir, \dots, o_n:dir)/c}$) adds a new column r to the input relation and assigns the row number to each row. The row number value is determined by the position of the row with respect to the given order $b_1:dir, \dots, b_n:dir$ where b_n defines the column to sort by and dir denotes the direction (ascending or descending) and a possible grouping c .

3 Target System

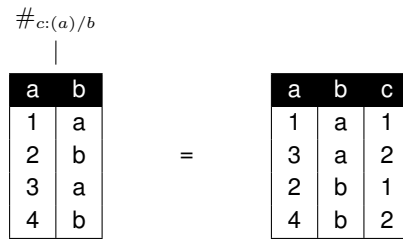


Figure 3.17: Example: # operator

Rowrank

The rowrank operator ($rnk_{r:(o_1, \dots, o_n)}$) adds a new column r to the input relation and assigns an ordinal value to each row. The ordinal value is determined by the position of the row with respect to the given order ($o_1:dir, \dots, o_n:dir$), where o_n denotes the column to be sorted by and dir specifies the direction (ascending or descending). If multiple rows contain the same value, they are assigned the same position value.

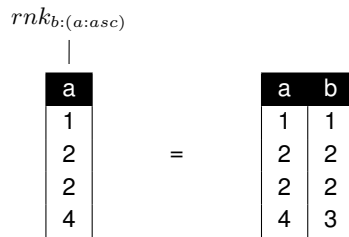
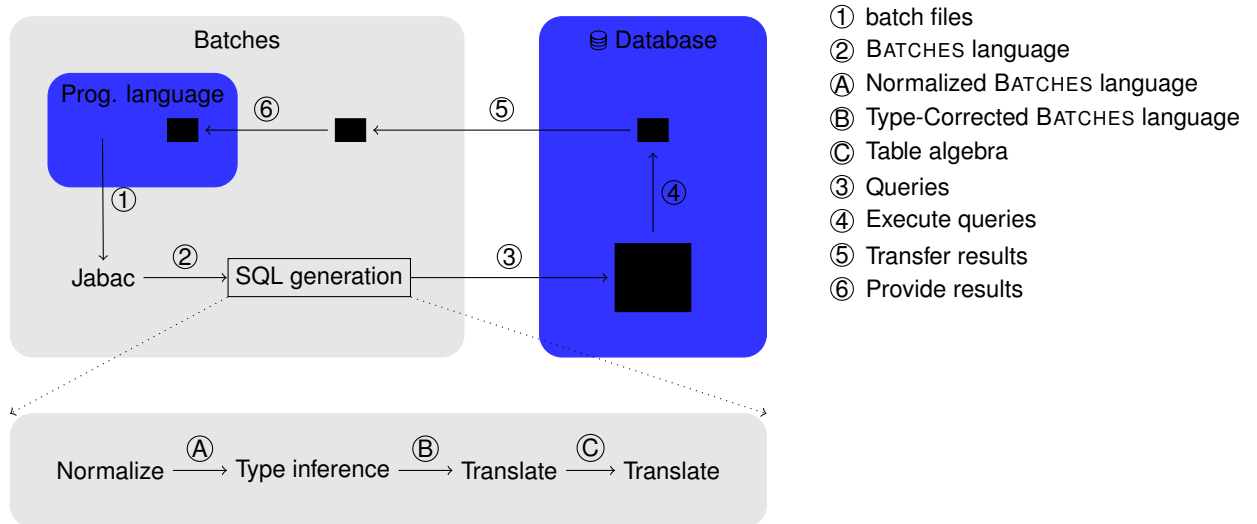


Figure 3.18: Example: rnk operator



4 Translation

The input language represented by *ASTs* generated by *Jabac* serves as input for our FERRY based translation to a target query language. The major steps processed are:

1. Normalization of the input language
2. Type correction of the normalized input language
3. Translate input language to intermediate language
4. Translate intermediate language to target query language
5. Write results to the *Forest*

The following sections will give a detailed description of what happens in every step of the translation performed.

4.1 Introduction

4.1.1 Notation

To describe the translation process we use inference rules of the form

4 Translation

$$\frac{\begin{array}{c} \textit{premis1} \\ \dots \\ \textit{premisn} \end{array}}{\Gamma; \textit{loop} \vdash e \rightarrow (q, \textit{cols}, \textit{itbls})}$$

where the premises together form the translation function \rightarrow of a rule. So the premises can be thought of as substeps of the translation of an expression e (and its possible subexpressions). The result consists of the following elements:

- $(q, \textit{cols}, \textit{itbls}, \textit{outs})$

The triple $(q, \textit{cols}, \textit{itbls})$ forms a so called queryinformationnode. A queryinformationnode contains a constructed algebraic plan q , a map \textit{cols} serving as a mapping of column names to concrete columns ($\{\dots, \text{"col1"} \rightarrow \text{column}, \dots\}$) and a map \textit{itbls} that contains columns mapping to queryinformationnodes ($\{\dots, \text{column} \rightarrow \text{qi}, \dots\}$). As the in \textit{itbls} referenced queryinformationnodes on their part can reference other queryinformationnodes the arising structure is a tree. Generally speaking q represents the outer table and the referenced queryinformationnodes contained in \textit{itbls} serve as inner tables. The \textit{outs} component actually stores the outputs to be able to fill the *forest* later on. The \textit{outs} component is represented by a map that associates variable names with a list ($\{x \rightarrow []\}$) containing output names. As outputs are passed through the whole compilation process, they will not be mentioned explicitly in every compilation rule. It will only appear when explicitly used by a rule (such as the compilation of e^l).

- Γ

Denotes the current variable environment containing names that map to already constructed queryinformationnodes ($\{\dots, x \rightarrow \text{qi}, \dots\}$). Note that not every queryinformationnode put in the environment is visible in every scope.

- \textit{loop}

Describes the current iteration context and as of that is needed to create loop lifted representations.

4.1.2 Handling Inner Tables

Adhere Inner Tables

Some translation rules adhere (e.g. union) outer tables - as a consequence we need to adhere inner tables as well. As inner tables on their part can contain inner

4 Translation

tables we need to process the adherence recursively. As defined in [7], we use the notation

$$q_o \vdash (itbls_{o_1}, itbls_{o_2}) \xrightarrow{app} itbls_o$$

describing that we based on an algebraic plan q_o the given maps $itbls_{o_1}$ and $itbls_{o_2}$ are combined to a single map $itbls_o$. The map $itbls_{o_1}$ belongs to an outer table O_1 and $itbls_{o_2}$ belongs to an outer table O_2 which already were adhered as exemplarily shown in Figure 4.1 (a). As result of the adherence of O_1 and O_2 , q_o was generated and passed to our rule. The generated plan q_o contains the old surrogate keys of the outer tables *item1* as well as the generated new surrogate keys *item'* and the proper order of the rows *item0*. Now we need to adhere the inner tables using the rule \xrightarrow{app} as well. The rule

$$\begin{array}{c}
 \textcircled{1} \quad q \equiv \#_{item':(iter1:asc,item0:asc,pos1:asc)}(@_{item0:1}(q_1) \cup @_{item0:2}(q_2)) \\
 q' \equiv (q) \bowtie_{item0=item0' \wedge iter=iterx'} (\prod_{item0':item0,item'':item',itemx':itemx}(q_0)) \\
 \textcircled{2} \quad q'' \equiv \prod_{iter:item'',pos,cols \setminus keys(itbls_1),keys(itbls_1):item'}(q') \\
 \textcircled{3} \quad q \vdash (itbls_1, itbls_2) \xrightarrow{app} itbls' \quad \textcircled{4} \quad q_0 \vdash (itbls_y, itbls_z) \xrightarrow{app} itbls'' \\
 \hline
 q \vdash ([itemx \rightarrow (q_1, cols_1, itbls_1)] + itbls_y, itemx \rightarrow (q_2, cols_2, itbls_2)] + itbls_z) \\
 \xrightarrow{app} [itemx \rightarrow (q'', cols_1, itbls')] + itbls''
 \end{array} \quad (APP)$$

describes the recursive adherence of inner tables. First of (①) all two corresponding inner tables (respectively algebraic plans) are determined. Two inner tables are considered corresponding when they are referenced by the same column in the passed surrogates maps. Above q_1 and q_2 were selected. To both, q_1 and q_2 we append a column *item0* containing value 1 for q_1 and value 2 for q_2 to preserve the correct order between the containments of the two plans. That followed q_1 and q_2 are united and the rownumber operator is applied to construct new unique surrogate keys stored in *item'* in case the inner tables q_1 and q_2 contain inner tables themselves. Without generating new surrogate keys, we would have the problem that both tables use the same old surrogate keys - so the mapping would not be correct. The generated new surrogate keys for possible inner tables of q_1 and q_2 are displayed in orange in Figure 4.2. The resulting plan q (Figure 4.2) now contains the old surrogate keys *iter*, the new surrogate keys *item'* for possible inner tables as well as the order information *item0* the positions of the values (*pos*) and the concrete values of the items (*item1*).

4 Translation

After having united the inner plans q_1 and q_2 we need to adapt the surrogate keys (②) of the outer and inner tables. To do that, we use the passed plan q_0 that contains the new surrogate keys ($item'$, after projection $item''$) as well as the old ones ($item1$, after projection $item1'$) and the order information ($item0$, after projection $item0'$). By joining q and the projected version of q_0 , we relate the rows using the old surrogate keys (condition: $iter = item1'$) and the order (condition: $item0 = item0'$). The new surrogate keys are mapped to the correct items this way. Now the old surrogate keys can be discarded and the new surrogate keys are taken over by renaming $item''$ to $iter$.

As q_1 as well as q_2 can contain inner tables on their part, we need to recurse into depth (③) using the same approach as explained above. The generated plan q would be passed to a new call to \xrightarrow{app} as q already contains the new surrogate keys ($item'$).

Additionally the outer tables could contain further surrogates besides q_1 and q_2 . To handle this we call \xrightarrow{app} recursively (④) for the possibly remaining surrogates of the outer tables.

The rule \xrightarrow{app} terminated, if both all surrogates from the outer table and all surrogates of inner tables have been adhered.

O_1		
iter	pos	item
1	1	1
2	1	2

I_1		
iter	pos	item
1	1	3
1	1	4
2	1	3
2	1	4

O_2		
iter	pos	item
1	1	1
2	1	2

I_2		
iter	pos	item
1	1	5
1	1	6
2	1	5
2	1	6

(a) Outer table O_1 with its inner table I_1

(b) Outer table O_2 with its inner table I_2

$O_1 \cup O_2$		
iter	pos	item
1	1	1
1	2	2
2	1	3
2	2	4

$I_1 \cup I_2$		
iter	pos	item
1	1	3
1	2	4
2	1	5
2	2	6
3	1	3
3	2	4
4	1	5
4	2	6

(c) Adhered tables O_1, O_2 and I_1, I_2

Figure 4.1: Example: Adherence of outer and inner tables

4 Translation

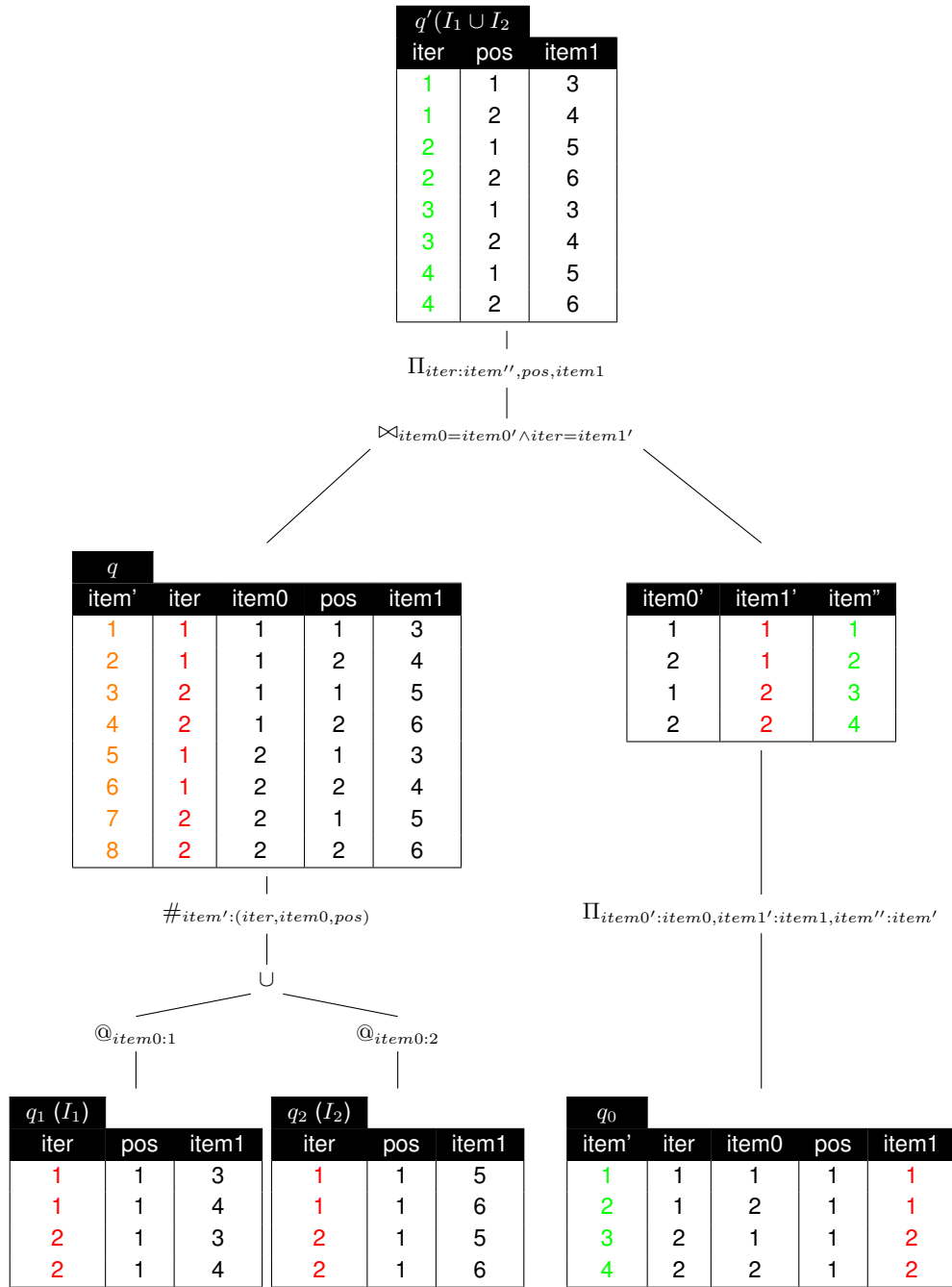


Figure 4.2: Example: Adherence of inner tables

Filter Inner Tables

Some translation rules (e.g. where) remove rows from tables. The removed lines in the outer table have to be removed in the inner tables as well. Again, as the inner tables may contain inner tables on their part we are forced to remove lines throughout the query information node tree recursively. As defined in [7], we use the

4 Translation

notation

$$q_o \vdash itbls \xrightarrow{sel} itbls'$$

describing that we, given q_0 , recursively remove the affected lines from the inner tables and as a result receive $itbls'$, only containing the remaining lines. The rule

$$\frac{\begin{array}{l} \textcircled{1} q' \equiv \Pi_{iter,pos,cols}(\Pi_{surr':surr}((q_0)) \bowtie_{surr'=iter} (q)) \\ \textcircled{2} q \vdash itbls \xrightarrow{sel} itbls' \quad \textcircled{3} q_0 \vdash itbls_y \xrightarrow{sel} itbls'' \end{array}}{q_0 \vdash [surr \rightarrow (q, cols, itbls)] + itbls_y \xrightarrow{sel} [surr \rightarrow (q', cols, itbls')] + itbls''} \quad (\text{SEL})$$

describes the recursive removal of affected lines of inner tables. First of all an inner table from the given $itbls$ component is chosen, subsequently called q . Now the passed algebraic plan q_0 , in which we already have removed the affected lines, is used to discard lines from the inner table q by applying an equi join to q and q_0 with the condition that the surrogate keys have to be equal ($surr' = iter$) (①). Now the resulting plan q' only contains rows that are not affected by the operation.

In the following, we have to continue this approach for every inner table that possibly is located in the inner tables' (q) surrogate map - so we recurse into depth (②).

Moreover, the $itbls$ component of the outer table may, besides the column $surr$, contain further surrogates. For this reason we have to process the possible other surrogates as well. Again, we use recursion to discharge every single surrogate contained in the $itbls$ component (③).

Similar to the adherence of tables, the rule \xrightarrow{sel} terminates, if both, all surrogates from the outer table and all surrogates of inner tables, have been processed.

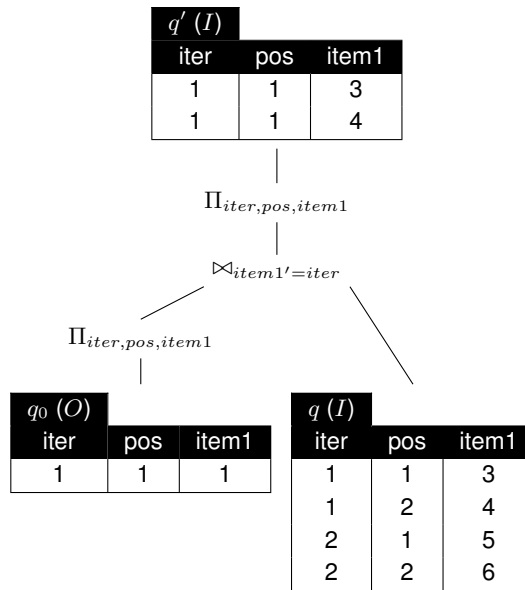
Figure 4.3 (a) illustrates the process of removing columns from outer and inner tables.

4 Translation

<i>O</i>		
iter	pos	item1
1	1	1
1	2	2

<i>I</i>		
iter	pos	item1
1	1	3
1	2	4
2	1	5
2	2	6

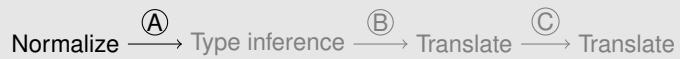
(a) Removing lines from an outer table (*O*) and its inner table (*I*)



(b) Process of removing columns given q_0

Figure 4.3: Example: Filter tables

4 Translation



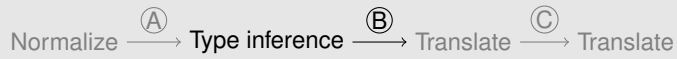
4.2 Normalize Input Tree

The first step in the translation from *AST* to the table algebra is a normalization of the tree that replaces constructs of the input language to be easier to handle later on. Easier to handle, in this case means, for example to clear weaknesses of the used visitor pattern when information of node is needed in a node situated above.

4.2.1 TableDot

To simplify the recognition whether a *e.f* expression represents the construction of a reference table or the access of an attribute, we replace constructs by a *TableDot* node that only holds the name of the database table to be referenced.

4 Translation



4.3 Type Inference

The type inference analyses *Jabac*'s generated *AST* with respect to introduce necessary *box* and *unbox* expressions. The expected implementation types *Row* and *Table* are defined for every expression occurring in the *AST* as they may differ due to the operations used. Subsequently, we check if every expression is of the implementation type expected by its parent expression - if so nothing is to do - thus no call to *box/unbox* is introduced. Deflecting types have to be considered by introducing the necessary expression. The function *resolve*

$$\text{resolve}(exp, act, e) = \begin{cases} \mathbf{box}(e) & exp = Row, act = Table \\ \mathbf{unbox}(e) & exp = Table, act = Row \\ e & \end{cases}$$

handles decision whether to introduce *box* or *unbox* expressions or not. The argument *exp* describes the expected implementation type, *act* describes the given implementation type of expression *e*.

The following inference rules of the form

$$\frac{\begin{array}{c} \textit{premis1} \\ \dots \\ \textit{premisn} \end{array}}{\textit{consequence}}$$

describe which implementation type we expect of the given arguments (premises) and the implementation type of the consequence. If any of the expected implementation types differs from what we expect, the function *resolve* corrects the type.

$$\frac{e_1, \dots, e_n :: Row}{[e_1, \dots, e_n] :: Table} \quad (\text{LIST})$$

4 Translation

$$\frac{e_1, \dots, e_n :: Row}{(e_1, \dots, e_n) :: Row} \quad (\text{RECORD})$$

The *for* expression expects a expression e_1 of type *Table*, a variable x of type *Row* to bind values of to collection e_1 to and a expression e_2 of type *Row*. After the loop has been processed, we expect it to return with type *Table*.

$$\frac{e_1 :: Table \quad x :: Row \quad e_2 :: Row}{\mathbf{forSEQ}(x \in e_1)e_2 :: Table} \quad (\text{FOR})$$

As *distinct* operates on an input set e_1 and removes every duplicate we expect it to return a set as well.

$$\frac{e_1 :: Table}{e_1.\mathit{distinct} :: Table} \quad (\text{DISTINCT})$$

Average operates on a set e_1 as input, applies the given function to each of its elements e_2 that have to be of type *Row*, finally sums up the function results and divides them through the count of values. So we expect the average function to return a single *Atom* value.

$$\frac{e_1 :: Table \quad e_2 :: Row}{e_1.\mathit{average}(\lambda x.e_2) :: Row} \quad (\text{AVERAGE})$$

Sum/dSum operate on a set e_1 and add the results of a function applied to every atomar value e_2 of the set. Finally a *Row* value is expected to be returned.

$$\frac{e_1 :: Table \quad e_2 :: Row}{e_1.\mathit{sum}(\lambda x.e_2) :: Row} \quad (\text{SUM/DSUM})$$

Min/dMin operate on a set e_1 and compare the results of a function applied to every atomar value e_2 of the set in order to determine the lowest one. Finally a *Row* value is expected to be returned.

4 Translation

$$\frac{e_1 :: Table \quad e_2 :: Row}{e_1.min(\lambda x.e_2) :: Row} \quad (\text{MIN/DMIN})$$

Max/dMax operate on a set e_1 and compare the results of an function applied to every atomar value e_2 of the set in order to determine the highest one. Finally a *Row* value is expected to be returned.

$$\frac{e_1 :: Table \quad e_2 :: Row}{e_1.max(\lambda x.e_2) :: Row} \quad (\text{MAX/DMAX})$$

First operates on a set e_1 and prunes it to the count of elements the atomar value n defines. So *first* is expected to return an object of type *Table*.

$$\frac{e_1 :: Table \quad n :: Row}{e_1.first(n) :: Table} \quad (\text{FIRST})$$

Exists operates on a set e_1 , checks if the set is empty and is expected to return an *Atom* value. There are two shapes of the *exists* function sharing this type inference.

$$\frac{e_1 :: Table}{e_1.exists() :: Row} \quad (\text{EXISTS})$$

$$\frac{e_1 :: Table \quad e_2 :: Row}{e_1.exists(\lambda x.e_2) :: Row} \quad (\text{EXISTS}(\lambda x.e))$$

All operates on a set e_1 and checks if the given function is satisfied by every single *Atom* e_2 of the set.

$$\frac{e_1 :: Table \quad e_2 :: Row}{e_1.all(\lambda x.e_2) :: Row} \quad (\text{ALL})$$

Count operates on a set e_1 and counts the *Atoms* e_2 the given function is satisfied. There are two different shapes of this operator sharing this type inference.

4 Translation

$$\frac{e_1 :: Table}{e_1.count() :: Row} \quad (\text{COUNT})$$

$$\frac{e_1 :: Table \quad e_2 :: Row}{e_1.count(\lambda x.e_2) :: Row} \quad (\text{COUNT}(\lambda x.e))$$

Where operates on a set e_1 and selects all items e_2 that satisfy the given function. So it is expected to return a set of type *Table*.

$$\frac{e_1 :: Table \quad e_2 :: Row}{e_1.where(\lambda x.e_2) :: Table} \quad (\text{WHERE})$$

OrderBy operates on a set e_1 and orders all items according to their attribute returned by the function applied on the elements e_2 . Whether to order ascending or descending depends on the given *Row* b . So it is expected to return a set of type *Table*.

$$\frac{e_1 :: Table \quad e_2 :: Row \quad b :: Row}{e_1.orderBy(\lambda x.e_2, b) :: Table} \quad (\text{ORDERBY})$$

GroupBy operates on a set e_1 and groups all items according to their attribute returned by the function applied on every single item e_2 . So the expected return type will be *Table*.

$$\frac{e_1 :: Table \quad e_2 :: Row}{e_1.groupBy(\lambda x.e_2) :: Table} \quad (\text{GROUPBY})$$

Project operates on a set e_1 and restricts all items on the attribute returned by the function applied on every single item e_2 . So the expected return type will be *Table*.

$$\frac{e_1 :: Table \quad e_2 :: Row}{e_1.project(\lambda x.e_2) :: Table} \quad (\text{PROJECT})$$

4 Translation

Union operates on a set e_1 and merges it with a set e_2 . So the expected return type will be *Table*.

$$\frac{e_1 :: Table \quad e_2 :: Table}{e_1.union(e_2) :: Table} \quad (\text{UNION})$$

Intersect operates on a set e_1 and intersects it with a set e_2 . So the expected return type will be *Table*.

$$\frac{e_1 :: Table \quad e_2 :: Table}{e_1.instersect(e_2) :: Table} \quad (\text{INTERSECT})$$

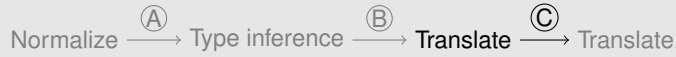
Binary operations take two arguments of type *Row*, apply the function given and return the result of the function as a single *Row*.

$$\frac{e_1 :: Row \quad e_2 :: Row}{\odot(e_1, e_2) :: Row} \quad (\text{BINARY OPERATION})$$

Unary operations take one arguments of type *Row*, apply the function given and return the result of the function as a single *Row*.

$$\frac{e_1 :: Row}{\odot(e_1) :: Row} \quad (\text{UNARY OPERATION})$$

4 Translation



4.4 Translate to Table Algebra

The translation to table algebra is done by compositional rules - that means that for every expression of the BATCHES language a rule was implemented which handles the specific translation and is completely independent from any other rule.

The implemented rules are introduced subsequently.

Output

To translate outputs e^l we first translate e . In e a variable will appear that identifies a entry in the environment (here x). That variable will be stored and put to the outputs of the queryinformationnode associated with its name l . Every time an access to the same variable is made, the name l of that output will be added to the list referenced by the variable x .

$$\frac{[\dots, x \rightarrow qi_x, \dots]; loop \vdash e \rightarrow ((q, cols, itbls), x)}{[\dots, x \rightarrow qi_x, \dots]; loop \vdash e^l \rightarrow (q, cols, itbls, \{x \rightarrow [l]\})} \quad (\text{OUTPUT})$$

Record

Records are translated by first translating the first component e_1 of the record. The rest of the record (e_2, \dots, e_n) first forms a new record and is processed afterwards step by step. As we are going to join q_{e_1} and q_{e_2} we need to adapt the names of the columns contained in the columnstructure $cols_{e_2}$. This is done by shifting the items numbers of $cols_{e_2}$ by the cardinality $|cols_{e_1}|$ of the expression translated before (e_1). For example: If $cols_{e_1}$ contains the columns $item1$ and $item2$ the cardinality $|cols_{e_1}|$ would be 2. So if $cols_{e_2}$ contains the columns $item1$ and $item2$ as well, the result of the function *shift* would be $cols'_{e_2}$ containing the columns $item3$ and $item4$. The function *shift* is applied to prevent name clashes in the join. As a result we have to adapt the columns referencing inner tables in the $itbls_{e_2}$ as well, if any. So we have to apply the *shift* function for the items in the $itbls_{e_2}$ component

4 Translation

as well. After renaming the column $iter1$ to $iter2$ and the columns of the $cols_{e_2}$ to their shifted representations $cols'_{e_2}$ we join the plans q_{e_2} and q_{e_1} using their $iter$ columns. Finally we restore the $iter1, pos1$ $item$ schema by using another project operation. The resulting queryinformationnode will contain the plan q , the columns of the component $cols_{e_1}$ combined with the shifted columns of $cols_{e_2}$ as well as the $itbls_{e_1}$ component combined with the shifted $itbls'_{e_2}$ component. As this rule works recursive, it terminated when all expressions in the record to be translated, were processed.

$$\begin{array}{c}
 \Gamma; loop \vdash e_1 \rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1}) \quad \Gamma; loop \vdash e_2 \rightarrow (q_{e_2}, cols_{e_2}, itbls_{e_2}) \\
 cols'_{e_2} = shift(cols_{e_2}, |cols_{e_1}|) \quad itbls'_{e_2} = shift(itbls_{e_2}, |cols_{e_1}|) \\
 q \equiv \Pi_{iter1, pos1, cols_{e_1}, cols'_{e_2}} ((q_{e_1}) \bowtie_{iter1=iter2} (\Pi_{iter2:iter1, cols'_{e_2}:cols_{e_2}} (q_{e_2}))) \\
 \hline
 \Gamma; loop \vdash (e_1, e_2, \dots, e_n) \rightarrow (q, cols_{e_1} + cols'_{e_2}, itbls_{e_1} + itbls'_{e_2}) \quad (\text{RECORD})
 \end{array}$$

Let

Let is translated by first compiling e_1 , writing e_1 to the actual environment bound to the variable given (here: x). As a next step, we compile e_2 using the new environment including x . The queryinformationnode resulting from the compilation of e_2 is the overall result of the compilation.

$$\frac{\Gamma; loop \vdash e_1 \rightarrow qi_{e_1} \quad \Gamma + [x \rightarrow qi_{e_1}]; loop \vdash e_2 \rightarrow qi_{e_2}}{\Gamma; loop \vdash \text{let } x = e_1 \ e_2 \rightarrow qi_{e_2}} \quad (\text{LET})$$

OrderBy

Translating `orderBy` is done by first translating e_1 which delivers the queryinformationnode which is bound to the variable (here: x) in the environment. Subsequently x serves as the variable e_2 is invoked on, so the translation of e_2 is done in context of e_2 's translation. To prepare the invocation of a theta join merging the resulting plans q_{e_1} and q_{e_2} in order to have the column to sort by available as well as align the data of the two plans due to their pos and $iter$ columns, we rename the $pos1$ and $iter1$ columns of q_{e_2} to $pos2$ and $iter2$ to prevent name collisions when joining. The sort column $item_x$ is renamed to $item_y$, where y denotes the maximum $item$ number of the column structure of qi_{e_1} ($cols_{e_1}$) incremented by 1. Now we apply the $\#$ operator that first of all sorts the rows of the table by the value of the column $item_y$ in the direction prescribed by d . The function

4 Translation

$$dir(d) = \begin{cases} \mathbf{asc} & d = true \\ \mathbf{dsc} & d = false \end{cases}$$

determines whether to sort *ascending* or *descending* from the given boolean value d . If d is true, we sort *ascending*, otherwise the direction to use is *descending*. Based on that new order, the $\#$ operator calculates the new positional information for every row and stores it in a newly generated column $pos3$.

Finally we use `project` to rename column $pos3$ to $pos1$ and preserve the $iter1$ column as well as the columns contained in $cols_{e_1}$.

$$\begin{aligned} & \Gamma; loop \vdash e_1 \rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1}) = qi_{e_1} \\ & \Gamma \cup [x \rightarrow qi_{e_1}]; loop \vdash \lambda x.e_2 \rightarrow (q_{e_2}, \{item_x\}, itbls_{e_2}) \\ & q' \equiv (\Pi_{cols_{e_1}.max+1:item_x,pos2:pos1,iter2:iter1}(q_{e_2})) \\ & q'' \equiv ((q') \bowtie_{iter2=iter1 \wedge pos2=pos1}(q_{e_1})) \\ & \frac{q''' \equiv (\Pi_{cols_{e_1}:cols_{e_1},pos1:pos3,iter1:iter1}(\#_{pos3:(cols_{e_1}.max+1:dir(d))/iter1}(q'')))}{\Gamma; loop \vdash e_1.orderBy(\lambda x.e_2, d) \rightarrow (q''', cols_{e_1}, itbls_{e_1})} \quad (\text{ORDERBY}) \end{aligned}$$

In Figure 4.4 an example is given how the *orderby* operator is translated.

For

We translate e_1 (①) and use the resulting plan q_{e_1} to create a map of the plan that temporarily stores the $iter$ and pos values of q_{e_1} in $iter3$ and $pos2$ and holds a new iteration context ($iter2$) generated by the `rownum` operator applied to q_{e_1} with the order of $iter1$ and $pos1$ in ascending direction (②). Exemplarily the translation approach is shown in Figure 4.8.

Subsequently q' is generated by applying a `rownum` operation, similar to the one used for `map`, a projection renaming $iter2$ generated by the `rownum` operator to $iter1$ and preserving columns of $cols_{e_1}$, and an `attach` operator adding a column $pos1$ with value 1 (③). The generation of q' is illustrated in Figure 4.7 (a). Actually q' stores the columns of the columnstructure of e_1 to be able to process the data later on and because of that is put in the environment contained in a `queryinformationnode` storing $cols_{e_1}$ and $itbls_{e_1}$ as well.

In step ④ the containments of the current environment are lifted - that is joining every plan of every `queryinformationnode` in the environment with the created plan

4 Translation

$\Gamma; \begin{array}{|c|} \hline \text{iter1} \\ \hline 1 \\ \hline 2 \\ \hline \end{array}; ((a, 10), (b, 30), (c, 20)).\text{orderBy}(\lambda (UnitsInStock), \text{true}) \rightarrow$

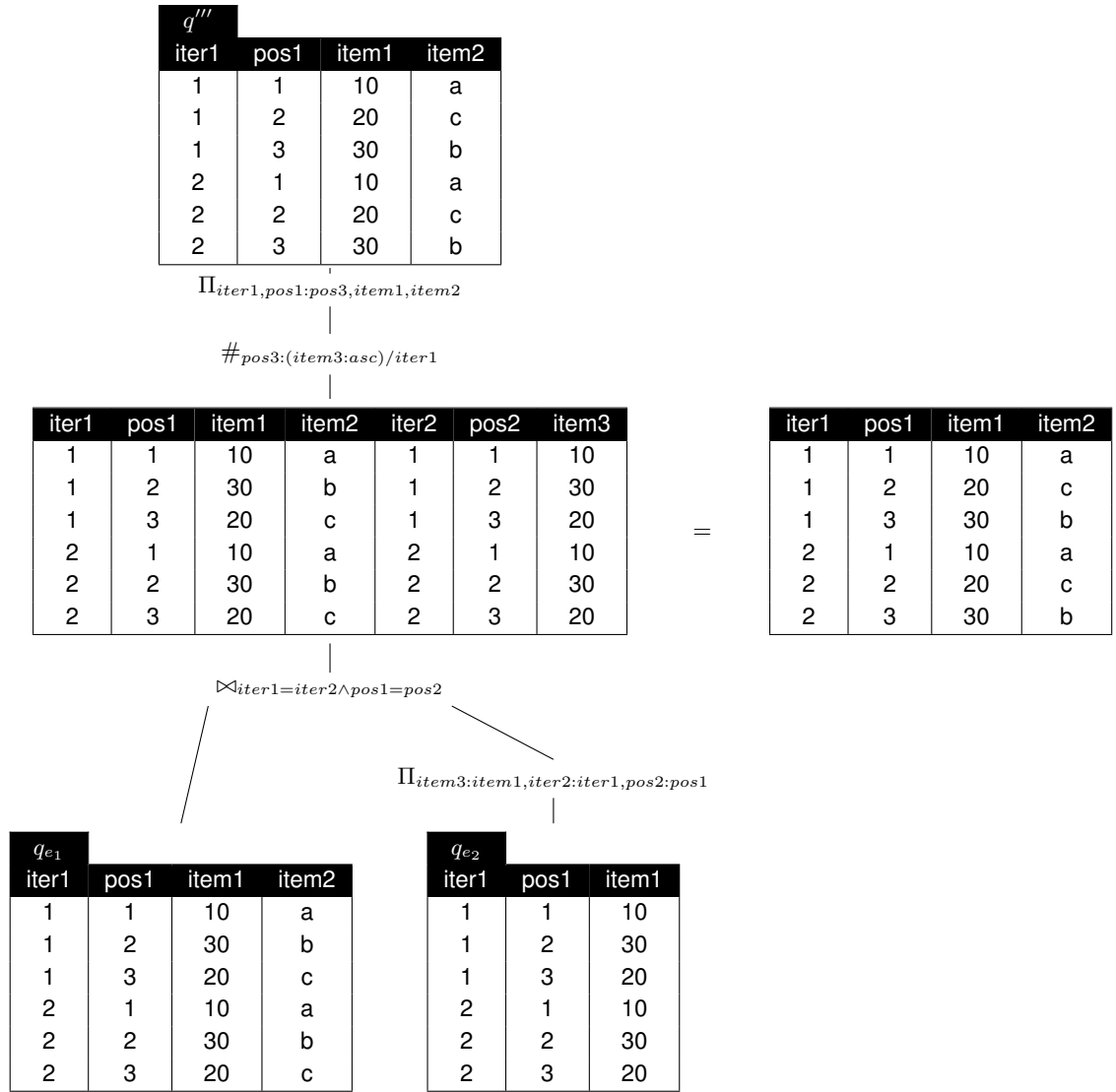


Figure 4.4: Example: OrderBy

map and exchanging their current $iter1$ column with the $iter2$ column of map . This approach is needed in order to have the correct iteration context for any further translation related to a loop.

Now we need to adapt the passed $loop$ context as well (5). To do that we use the generated plan q' and simply project out every column except $iter1$. The new $loop'$ now serves to translate any expression that is related to loop to be conform to the

4 Translation

lifted plans in the current environment. In Figure 4.7 (b) an example how the new *loop* context is created is given.

With the environment and the *loop* prepared in the way described above, we are now ready to translate e_2 . The resulting queryinformationnode will contain a plan q_{e_2} , the columns $cols_{e_2}$ and the surrogates $itbls_{e_2}$ (⑥).

Consequently, we have to restore the correct *iter1* and *pos1* columns of q_{e_2} . To do that in step ⑦ we join q_{e_2} with the created plan *map*, rename the *iter3* and *pos2* columns to *iter1* and *pos1* and preserve the *itemx* column in the resulting relation q''' .

The overall translation result now consists of a queryinformationnode holding plan q'' , the $cols_{e_2}$ component and $itbls_{e_2}$.

$$\begin{array}{l}
 \textcircled{1} \quad [\dots, qi_u \rightarrow (q_u, cols_u, itbls_u), \dots]; loop \vdash e_1 \rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1}) \\
 \textcircled{2} \quad map \equiv (\Pi_{pos2:pos1, iter2, iter3:iter1}(\#_{iter2:(iter1:asc, pos1:asc)}(q_{e_1}))) \\
 \textcircled{3} \quad q' \equiv (@_{pos1:1}(\Pi_{cols_{e_1}, iter1:iter2}(\#_{iter2:(iter1:asc, pos1:asc)}(q_{e_1})))) \\
 \textcircled{4} \quad \Gamma' \equiv [\dots, qi_u \rightarrow (\Pi_{iter1:iter2, pos1}(q_u \bowtie_{iter1=iter3} map)), cols_u, itbls_u, \dots] \\
 \textcircled{5} \quad loop' = (\Pi_{iter1}(q')) \\
 \textcircled{6} \quad \Gamma' \cup [v \rightarrow (q', cols_{e_1}, itbls_{e_1})]; loop' \vdash e_2 \rightarrow (q_{e_2}, cols_{e_2}, itbls_{e_2}) \\
 \textcircled{7} \quad q'' \equiv \Pi_{cols_{e_2}, iter1:iter3, pos1:pos2}(q_{e_2} \bowtie_{iter1=iter2} map) \\
 \hline
 [\dots, qi_u \rightarrow (q_u, cols_u, itbls_u), \dots]; loop \vdash ForSEQ(x \in e_1)e_2 \rightarrow (q'', cols_{e_2}, itbls_{e_2}) \quad (FOR)
 \end{array}$$

Count

As expressions of the form $e_1.count()$ are translated to **for COUNT**($x \in e_1$) e_2 we first of all have to translate the generated **for** expression in the way explained in Section 4.4. One may wonder why an expression of the form $e_1.count()$ is translated to a *for* expression holding two expressions. Actually, in this case e_2 represents the items to be processed by the count operator. The result of the translation of the *for* expression will be plan q'' which is used as base to apply the count of the elements in the list subsequently - so on q'' the count operator is applied. Count will deliver a relation which for every iteration holds the number of elements included in the list. To determine empty iterations the result of the count operation is projected to its column *iter1* and a difference operation is applied to it and the current loop context. The remaining items represent the iterations that do not contain any elements, so we attach a new column *item1* with the given value 0 to any row resulting from the difference operation. To combine the empty iterations

4 Translation

with the results of the count operation we apply a union operation to combine the two plans - so we get the count for every existing iteration (q''''). Finally, we add a new $pos1$ column with value 1 to result, as every row now contains a single value describing the count of elements in a iteration - so every count value is at position 1 in its iteration.

$$\begin{aligned}
 & [\dots, u \rightarrow (q_u, cols_u, itbls_u), \dots]; loop \vdash e_1 \rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1}) \\
 & map \equiv (\Pi_{pos2:pos1, iter2, iter3: iter1} (\#_{iter2: (iter1: asc, pos1: asc)} (q_{e_1}))) \\
 & q' \equiv (@_{pos1:1} (\Pi_{cols_{e_1}, iter1: iter2} (\#_{iter2: (iter1: asc, pos1: asc)} (q_{e_1})))) \\
 \Gamma' \equiv & [\dots, u \rightarrow (\Pi_{iter1: iter2, pos1} (q_u \bowtie_{iter1=iter3} map)), cols_u, itbls_u), \dots] \\
 & loop' = (\Pi_{iter1} (q')) \\
 \Gamma' \cup & [v \rightarrow (q', cols_{e_1}, itbls_{e_1}); loop' \vdash e_2 \rightarrow (q_{e_2}, cols_{e_2}, itbls_{e_2}) \\
 & q'' \equiv \Pi_{cols_{e_2}, iter1: iter3, pos1: pos2} (q_{e_2} \bowtie_{iter1=iter2} map) \\
 & q''' \equiv (count_{item1:()} / iter1 (q'')) \\
 q'''' \equiv & (@_{pos1:1} ((\Pi_{iter1, item1} (q''')) \cup (@_{item1:0} ((loop) \setminus (\Pi_{iter1} ((q''')))))))) \quad (\text{COUNT}()) \\
 & [\dots, q^i_u \rightarrow (q_u, cols_u, itbls_u), \dots]; loop \vdash e_1.count() \rightarrow (q''''', item1, [])
 \end{aligned}$$

The translation of the *count* operation first applying a function to items to determine which of them to count is translated in a very similar way. The first part of the translation again is similar to the translation of a *for* expression. The translation of e_2 in this case will deliver a relation that contains a single boolean column that delivers the results of the applied function $\lambda x.e_2$. Every row that satisfies the function will contain a value *true*, every row that does not satisfy the function will contain a value *false*. Now we are able to select the rows that satisfy the given function by applying the distinct operator to q_{e_2} . The resulting relation will only contain the rows that were evaluated to *true* when invoking the function. Subsequently the correct state of the duplicate free relation is reconstructed by joining it with the stored *map* relation and adding a project operation that takes over the *iter3* and *pos2* values of the *map* relation. The resulting relation q'' the count operator is now applied upon. For every iteration we count the number of rows as we already removed duplicates and store the result in *item1*. Finally we need to restore empty lists as well as lists that do not contain any values matching the given function, because these iterations were removed completely when applying the distinct operation. The approach to restore empty lists and lists containing not a single matching value is just the same as in the count operation without any arguments.

Both operations, *count* and *count*(λ) rely on the (un)boxing mechanism - so if we have to count the containments of a inner list a unbox call would have been introduced to be able to count the inner values of a nested list.

4 Translation

Figure 4.5 shows the translation process of $count(\lambda)$ without explicitly showing the translation of for . The restoring of empty iterations is shown exemplarily in Figure 4.6.

$$\begin{array}{l}
[\dots, u \rightarrow (q_u, cols_u, itbls_u), \dots]; loop \vdash e_1 \rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1}) \\
map \equiv (\Pi_{pos2:pos1, iter2, iter3:iter1}(\#_{iter2:(iter1:asc, pos1:asc)}(q_{e_1}))) \\
q' \equiv (@_{pos1:1}(\Pi_{cols_{e_1}, iter1:iter2}(\#_{iter2:(iter1:asc, pos1:asc)}(q_{e_1})))) \\
\Gamma' \equiv [\dots, u \rightarrow (\Pi_{iter1:iter2, pos1}(q_u \bowtie_{iter1=iter3} map)), cols_u, itbls_u, \dots] \\
loop' = (\Pi_{iter1}(q')) \\
\Gamma' \cup [v \rightarrow (q', cols_{e_1}, itbls_{e_1})]; loop' \vdash e_2 \rightarrow (q_{e_2}, cols_{e_2}, itbls_{e_2}) \\
q'' \equiv (\Pi_{pos1:pos2, iter1:iter3}((\sigma_{cols_{e_2}.max=true}(q_{e_2})) \bowtie_{iter1=iter2} (map))) \\
q''' \equiv (count_{item1:()}/iter1(q'')) \\
q'''' \equiv (@_{pos1:1}((\Pi_{iter1, item1}(q''')) \cup (@_{item1:0}((loop) \setminus (\Pi_{iter1}((q''')))))))) \\
\hline
\Gamma; loop \vdash e_1.count(\lambda x.e_2) \rightarrow (q''''', \{item1\}, []) \quad (COUNT(\lambda))
\end{array}$$

Average, Sum, Min, Max

Expressions of the form $e_1.agg(\lambda x.e_2)$ are translated to **for agg**($x \in e_1$) e_2 where $agg \in \{agg, sum, min, max\}$. So we first of all have to translate the generated **for** expression. We translate e_1 (①) and use the resulting plan q_{e_1} to create a map of the plan that temporarily stores the iter and pos values of q_{e_1} in $iter3$ and $pos2$ and holds a new iteration context ($iter2$) generated by the rownum operator applied to q_{e_1} with the order of $iter1$ and $pos1$ in ascending order (②). Exemplarily the translation approach is shown in Figure 4.8.

Then q' is generated by applying a rownum operation, similar to the one used for map, a projection renaming $iter2$ generated by the rownum operator to $iter1$ and preserving columns of $cols_{e_1}$, and an attach operator adding a column $pos1$ with value 1 (③). The generation of q' is illustrated in Figure 4.7 (a). Actually q' stores the columns of the columnstructure of e_1 to be able to process the data later on and because of that is put in the environment contained in a queryinformationnode storing $cols_{e_1}$ and $itbls_{e_1}$ as well.

In step ④ the containments of the current environment are lifted - that is joining every plan of every queryinformationnode in the environment with the created plan map and exchanging their current $iter1$ column with the $iter2$ column of map . This approach is needed in order to have the correct iteration context for any further translation related to a loop.

4 Translation

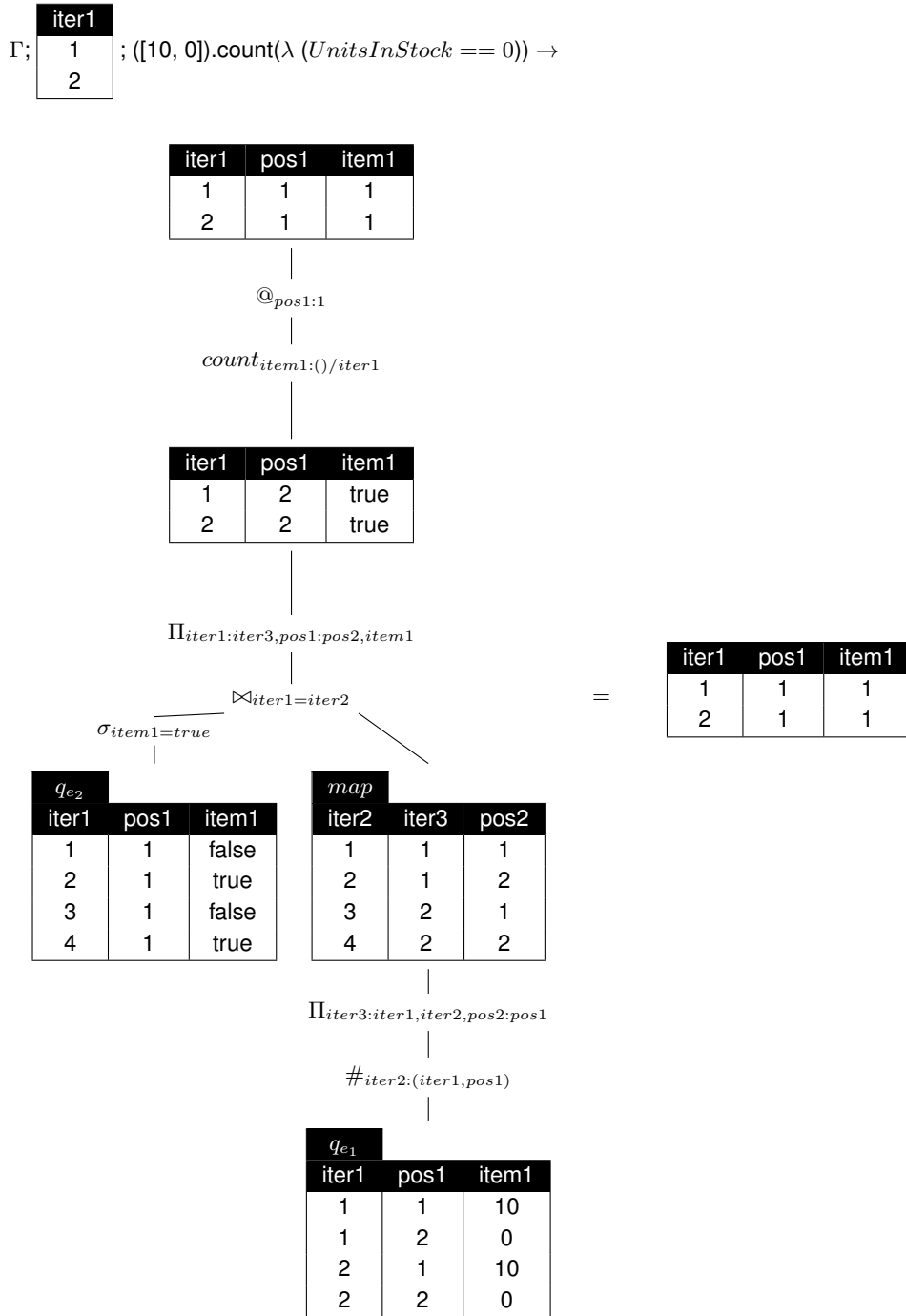


Figure 4.5: Example: count(λ)

Now we also need to adapt the passed *loop* context (5). To do so we use the generated plan q' and simply project out every column except *iter1*. The new *loop'* now serves to translate every expression that is related to loop in order to be conform to the lifted plans in the current environment. Figure 4.7 (b) shows an

4 Translation

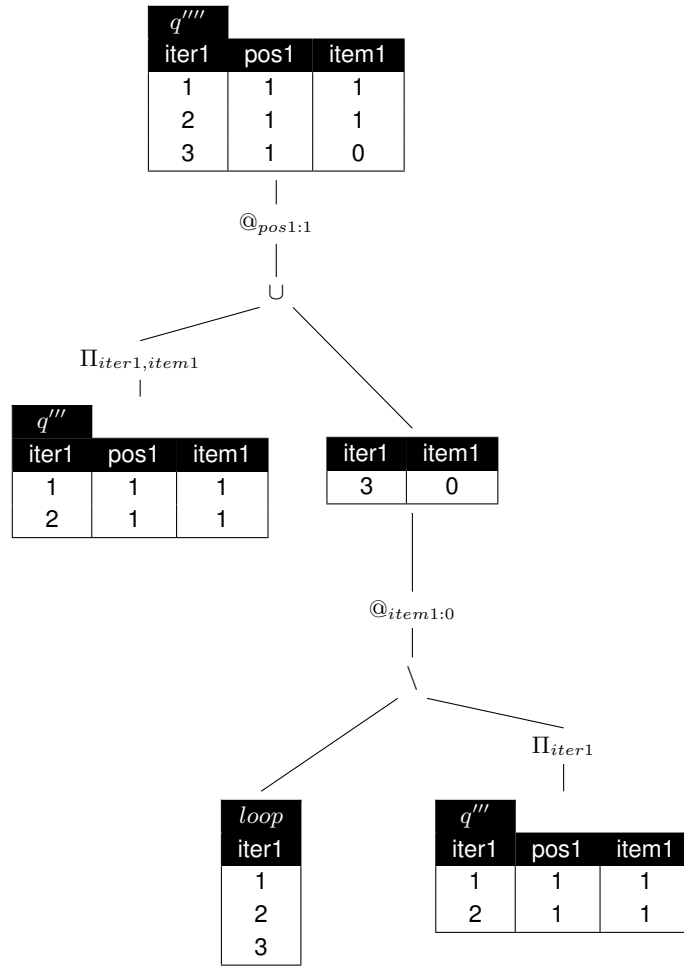


Figure 4.6: Example: Restoring empty iterations

example of how the new *loop* context is created is given.

With the environment and the *loop* prepared in the way described above, we are now ready to translate e_2 . The resulting queryinformationnode will contain plan q_{e_2} that encodes the result of the applied function with the *iter1* and *pos1* values generated for q' . The columnstructure will only contain a single *item* column containing the results of the function application. We do not expect the queryinformationnode to contain any surrogates as the compilation of a function will always contain the column *itemx* containing the results of the function application (⑥).

Afterwards, we have to restore the correct *iter1* and *pos1* columns of q_{e_2} . To do that in step ⑦ we join q_{e_2} with the created plan *map*, rename the *iter3* and *pos2* columns to *iter1* and *pos1* and preserve the *itemx* column in the resulting relation q''' . Having done so, we are now ready to apply the *agg* operation. The old column *itemx* is used to apply the aggregation on and serves as the column holding the

4 Translation

results of the aggregation as well. We calculate the aggregation partitioned by $iter1$, so the resulting relation contains exactly one result value for each iteration. Finally we attach a new $pos1$ column with value 1 as the resulting list, as mentioned above, contains a single value for every iteration only.

The overall translation result now consists of a queryinformationnode holding plan q''' , the single $itemx$ holding the results of the aggregation and no surrogates.

If the list we apply the agg function upon is nested, a $unbox$ expression would have been added in Type inference phase (Section 4.3).

$$\begin{array}{l}
\textcircled{1} \quad [\dots, u \rightarrow (q_u, cols_u, itbls_u), \dots]; loop \vdash e_1 \rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1}) \\
\textcircled{2} \quad map \equiv (\Pi_{pos2:pos1, iter2, iter3:iter1} (\#_{iter2:(iter1:asc, pos1:asc)} (q_{e_1}))) \\
\textcircled{3} \quad q' \equiv (@_{pos1:1} (\Pi_{cols_{e_1}, iter1:iter2} (\#_{iter2:(iter1:asc, pos1:asc)} (q_{e_1})))) \\
\textcircled{4} \quad \Gamma' \equiv [\dots, u \rightarrow (\Pi_{iter1:iter2, pos1} (q_u \bowtie_{iter1=iter3} map)), cols_u, itbls_u), \dots] \\
\textcircled{5} \quad loop' = (\Pi_{iter1} (q')) \\
\textcircled{6} \quad \Gamma' \cup [v \rightarrow (q', cols_{e_1}, itbls_{e_1})]; loop' \vdash e_2 \rightarrow (q_{e_2}, itemx, []) \\
\textcircled{7} \quad q'' \equiv \Pi_{itemx, iter1:iter3, pos1:pos2} (q_{e_2} \bowtie_{iter1=iter2} map) \\
\textcircled{8} \quad q''' \equiv (@_{pos1:1} (agg_{itemx:(itemx)/iter1} (q''))) \\
\hline
[\dots, q_{i_u} \rightarrow (q_u, cols_u, itbls_u), \dots]; loop \vdash e_1.agg(\lambda x.e_2) \rightarrow (q''', itemx, []) \quad (AGG)
\end{array}$$

All

Expressions of the form $e_1.all(\lambda x.e_2)$ are translated to **for all**($x \in e_1$) e_2 . So again, we first of all have to translate the generated **for** expression in the way explained in Section 4.4. The result will be plan q'' which is used as base to apply the *all* operation to, containing only one column $itemx$ storing the result of the function given to the *all* operation. Consequently, we apply the *all* operation to $itemx$ partitioned by $iter1$ to calculate if every item of an iteration satisfies the given function. The resulting plan q''' is now taken and a difference operation against loop is applied. By doing so we detect empty iterations and add a new column $itemx$ with the value *true* for every empty iteration in order not to distort the overall result of the *all* operation. This table of empty iterations is now united with the results of the *all* function (q''') and finally a new column $pos1$ with the given value 1 is attached to complete the *iter, pos item* demand. The value 1 is correct here for the $pos1$ column as every iteration in the resulting relation contains one row only.

If we have to operate on nested lists a *unbox* call would have been introduced to enable us to check the items of the inner list.

4 Translation

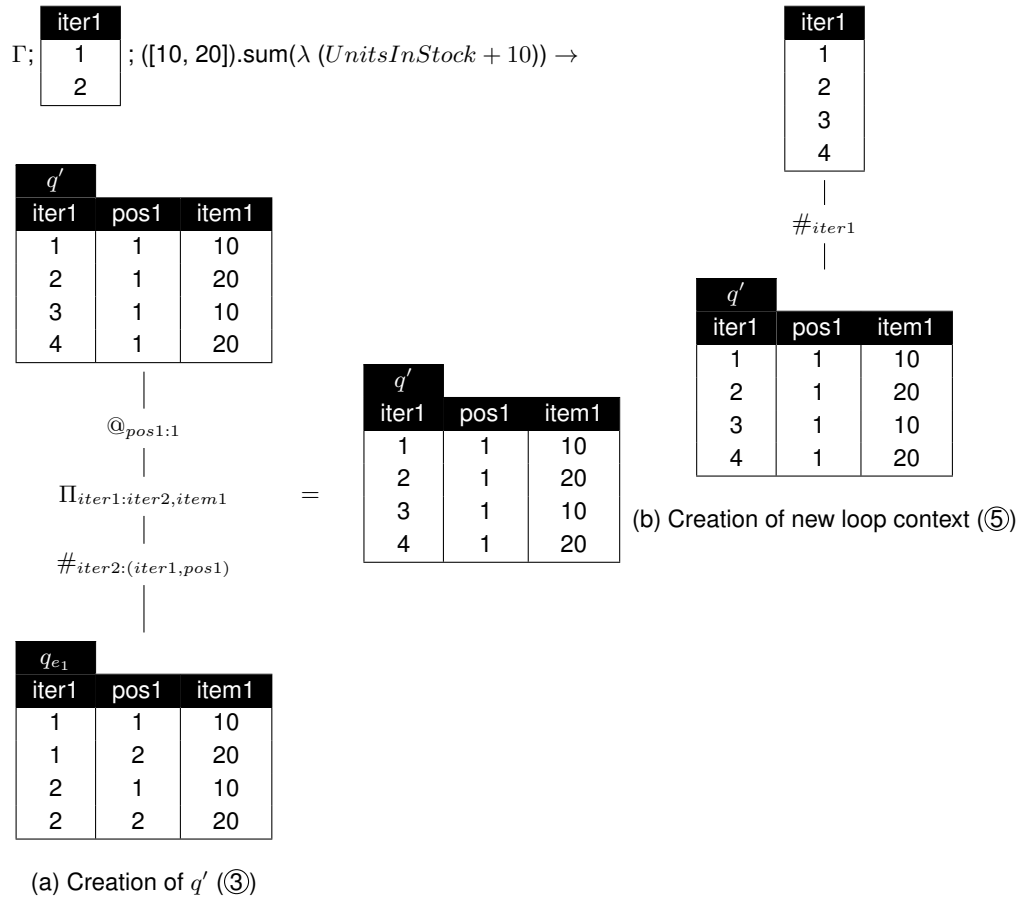


Figure 4.7: Example: Creation of q'

Figure 4.9 shows the translation of *all* exemplarily without the compilation of the **for** expression. The restoring of empty iterations is shown exemplarily in Figure 4.6.

4 Translation

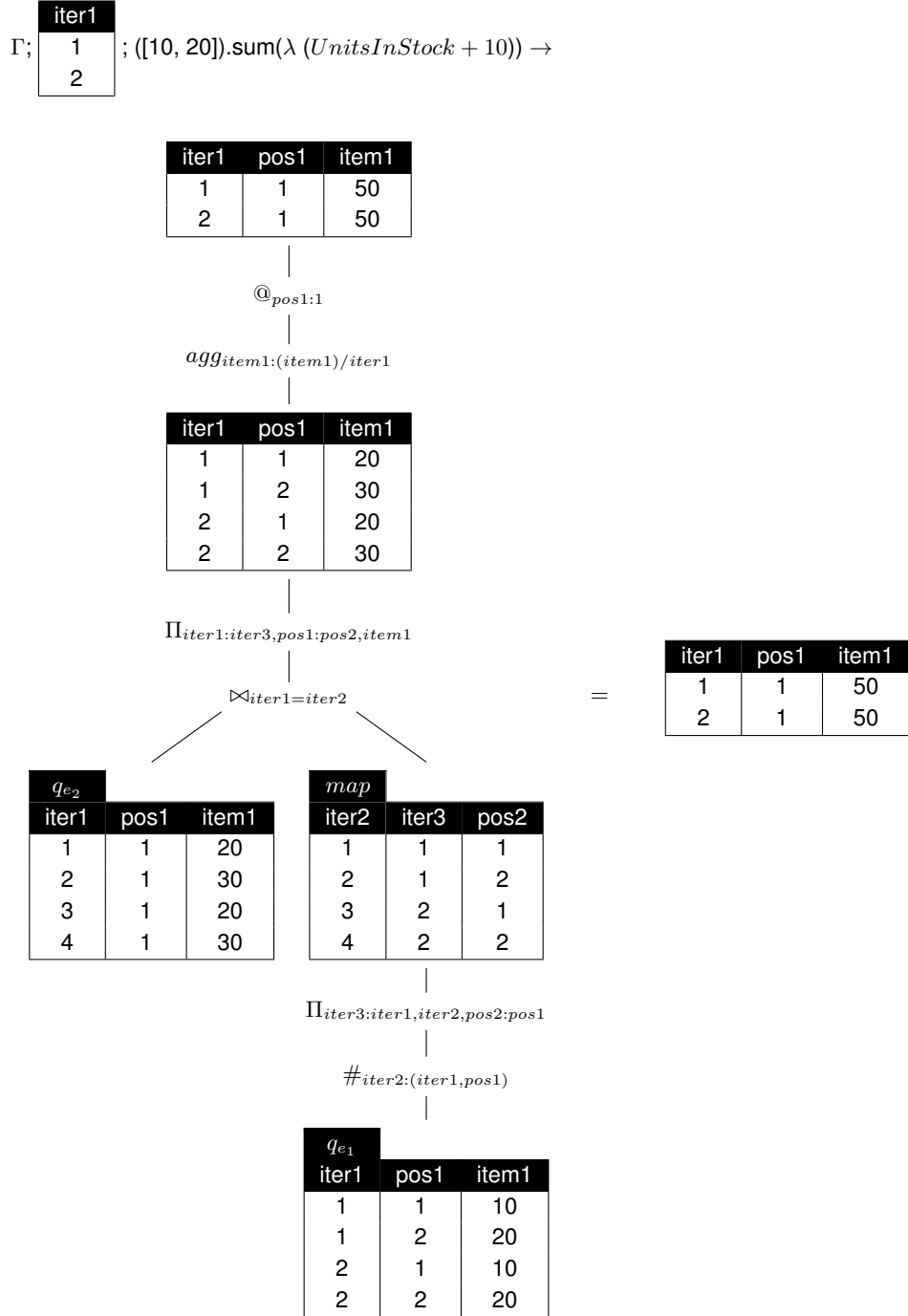


Figure 4.8: Example: sum

4 Translation

$$\begin{array}{c}
[\dots, qi_u \rightarrow (q_u, cols_u, itbls_u), \dots]; loop \vdash e_1 \rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1}) \\
map \equiv (\Pi_{pos2:pos1, iter2, iter3:iter1} (\#_{iter2:(iter1:asc, pos1:asc)}(q_{e_1}))) \\
q' \equiv (@_{pos1:1} (\Pi_{cols_{e_1}, iter1:iter2} (\#_{iter2:(iter1:asc, pos1:asc)}(q_{e_1})))) \\
\Gamma' \equiv [\dots, qi_u \rightarrow (\Pi_{iter1:iter2, pos1} (q_u \bowtie_{iter1=iter3} map)), cols_u, itbls_u, \dots] \\
loop' = (\Pi_{iter1}(q')) \quad \Gamma' \cup [v \rightarrow (q', cols_{e_1}, itbls_{e_1})]; loop' \vdash e_2 \rightarrow (q_{e_2}, itemx, []) \\
q'' \equiv \Pi_{itemx, iter1:iter3, pos1:pos2} (q_{e_2} \bowtie_{iter1=iter2} map) \\
q''' \equiv (all_{itemx:(itemx)/iter1}(q'')) \\
q'''' \equiv @_{pos1:1} (q''' \cup (@_{itemx:true} ((loop) \setminus (\Pi_{iter1}((q''')))))) \\
\hline
[\dots, qi_u \rightarrow (q_u, cols_u, itbls_u), \dots]; loop \vdash e_1.all(\lambda x.e_2) \rightarrow (q''''', itemx, []) \quad (ALL)
\end{array}$$

Exists

Exists is compiled the same way we compile *all* (Section 4.4). The only differences appearing are the used operator *exists* instead of *all* in q''' and the attached *itemx* in q''' which is added with value *false* instead of *true*. As we do not want to distort the overall result of the *exists* operation by empty iterations, we need to use *false* because otherwise q'''' would contain the value *true* for empty iterations even though no single element satisfies the given function.

Figure 4.9 shows the translation of *all* exemplarily without illustrating the merging of empty iterations and the compilation of the **for** expression and is very similar to the translation of the *exists* operator. The only thing to use the *exists* operator instead of the *all* operator. The restoring of empty iterations is shown exemplarily in Figure 4.6.

$$\begin{array}{c}
[\dots, qi_u \rightarrow (q_u, cols_u, itbls_u), \dots]; loop \vdash e_1 \rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1}) \\
map \equiv (\Pi_{pos2:pos1, iter2, iter3:iter1} (\#_{iter2:(iter1:asc, pos1:asc)}(q_{e_1}))) \\
q' \equiv (@_{pos1:1} (\Pi_{cols_{e_1}, iter1:iter2} (\#_{iter2:(iter1:asc, pos1:asc)}(q_{e_1})))) \\
\Gamma' \equiv [\dots, qi_u \rightarrow (\Pi_{iter1:iter2, pos1} (q_u \bowtie_{iter1=iter3} map)), cols_u, itbls_u, \dots] \\
loop' = (\Pi_{iter1}(q')) \quad \Gamma' \cup [v \rightarrow (q', cols_{e_1}, itbls_{e_1})]; loop' \vdash e_2 \rightarrow (q_{e_2}, itemx, []) \\
q'' \equiv \Pi_{itemx, iter1:iter3, pos1:pos2} (q_{e_2} \bowtie_{iter1=iter2} map) \\
q''' \equiv (exists_{itemx:(itemx)/iter1}(q'')) \\
q'''' \equiv @_{pos1:1} (q''' \cup (@_{itemx:false} ((loop) \setminus (\Pi_{iter1}((q''')))))) \\
\hline
[\dots, qi_u \rightarrow (q_u, cols_u, itbls_u), \dots]; loop \vdash e_1.exists(\lambda x.e_2) \rightarrow (q''''', itemx, []) \quad (EXISTS)
\end{array}$$

4 Translation

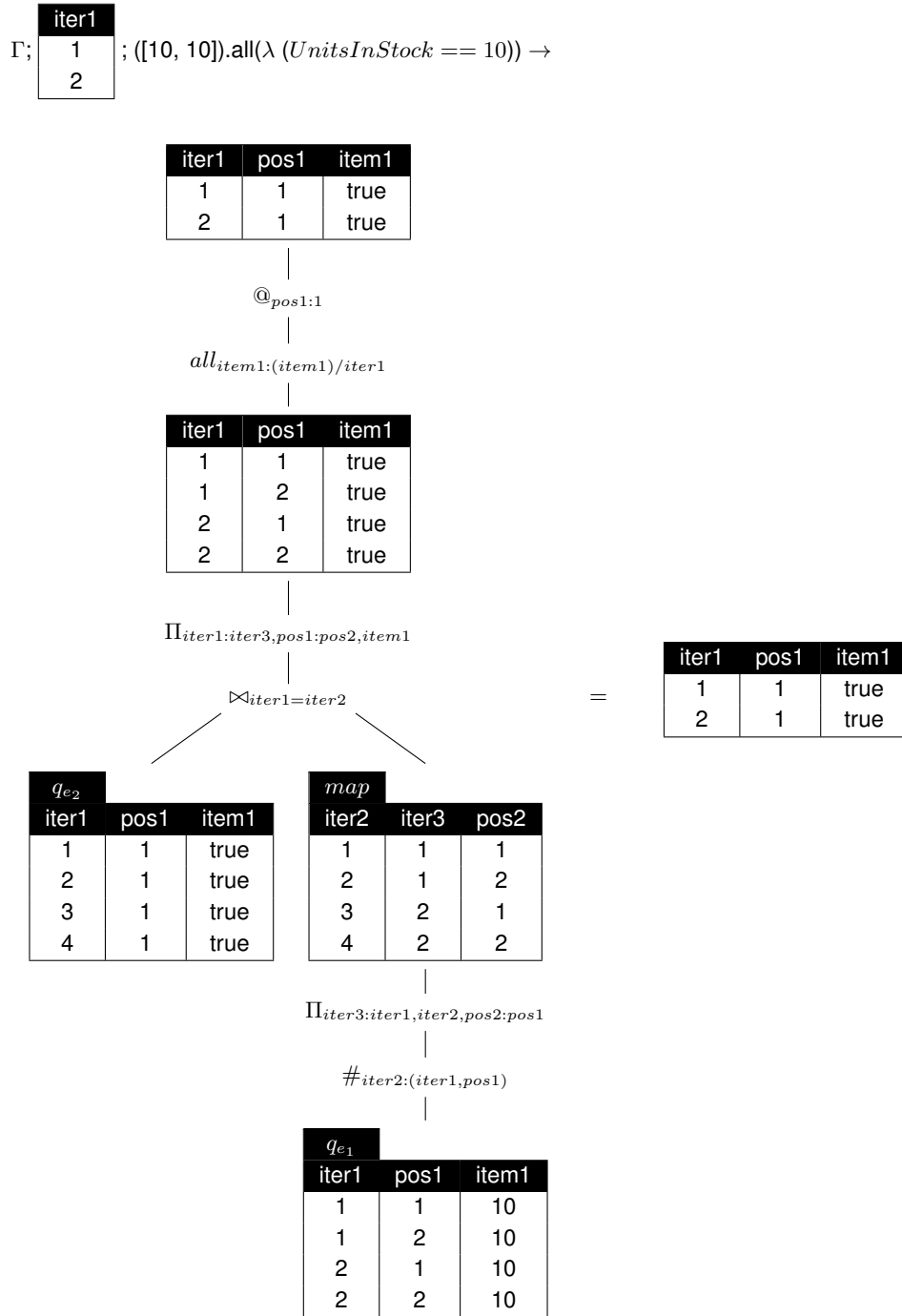


Figure 4.9: Example: all

Field access

Field accesses are realized by first compiling e_1 . The resulting query information node $q_{i_{e_1}}$ contains a column structure where names are mapped to columns. These names are used to access fields by looking up the column in the column structure

4 Translation

that is associated with the given name (here f). We then use the determined column to lookup possibly existing surrogates. Finally a project operation is used to select the $iter1$ and $pos1$ columns as well as the column determined by the given name f . So the resulting queryinformationnode is constructed by plan q , a column-structure containing the determined column and the surrogates are formed by the determined column and its associated queryinformationnode qi .

$$\begin{array}{c}
 \Gamma; loop \vdash e_1 \rightarrow (q_{e_1}, \{\dots, f \rightarrow column, \dots\}, [\dots, column \rightarrow qi, \dots]) \\
 \{\dots, f \rightarrow column, \dots\} \vdash f \rightarrow column \\
 \hline
 [\dots, column \rightarrow qi, \dots] \vdash column \rightarrow qi \quad q \equiv \Pi_{column, iter1, pos1}(q_{e_1}) \\
 \Gamma; loop \vdash e_1.f \rightarrow (q, column, [column \rightarrow qi])
 \end{array} \quad (\text{FIELDACCESS})$$

Project

Project is translated in the exactly the same manner as **for SEQ**. For the detailed translation see Section 4.4.

$$\begin{array}{c}
 [\dots, u \rightarrow (q_u, cols_u, itbls_u), \dots]; loop \vdash e_1 \rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1}) \\
 map \equiv (\Pi_{pos2:pos1, iter2, iter3:iter1}(\#_{iter2:(iter1:asc, pos1:asc)}(q_{e_1}))) \\
 q' \equiv (@_{pos1:1}(\Pi_{cols_{e_1}, iter1:iter2}(\#_{iter2:(iter1:asc, pos1:asc)}(q_{e_1})))) \\
 \Gamma' \equiv [\dots, u \rightarrow (\Pi_{iter1:iter2, pos1}(q_u \bowtie_{iter1=iter3} map)), cols_u, itbls_u), \dots] \\
 loop' = (\Pi_{iter1}(q')) \\
 \Gamma' \cup [v \rightarrow (q', cols_{e_1}, itbls_{e_1})]; loop' \vdash e_2 \rightarrow (q_{e_2}, cols_{e_2}, itbls_{e_2}) \\
 q'' \equiv \Pi_{cols_{e_2}, iter1:iter3, pos1:pos2}(q_{e_2} \bowtie_{iter1=iter2} map) \\
 \hline
 [\dots, u \rightarrow (q_u, cols_u, itbls_u), \dots]; loop \vdash e_1.project(\lambda x.e2) \rightarrow (q'', cols_{e_2}, itbls_{e_2})
 \end{array} \quad (\text{PROJECT})$$

First

To translate the *first* operator we primarily translate the expressions e_1 and e_2 where e_1 represents the list to apply *first* on and e_2 is a constant integer value that denotes the number of elements we want to extract from the list. As e_2 always has to be a constant integer value, the rule *constant* is applied here (see Section 4.4) - so the translation of e_2 always delivers the loop lifted representation of the constant with only one item in the columnstructure. Based on that information we use the project operator to rename the $iter1$ column to $iter2$ and the single item $item1$ to $itemx$ where x is the maximum column value contained in $cols_{e_1}$ incremented by

4 Translation

one in order to prevent name clashes when joining q_{e_1} and q_{e_2} . Now we join q_{e_1} and the projected plan of q_{e_2} to be able to select all rows where the value of column $pos1$ is lower than or equal to the position value contained in the $itemx$ column. The resulting relation will only contain the first n values where n denotes the value encoded by the expression e_2 . Finally we apply another project operator to restore the $iter, pos, item$ form of the plan and eliminate the unnecessary columns such as $itemx$.

If the queryinformationnode resulting from the translation of e_1 contains surrogate keys, we also have to select the n values in the inner plans. To do so, we use the rule \xrightarrow{sel} introduced in Section 4.1.2 which recursively selects the correct rows in the queryinformationnodes. Figure 4.10 exemplary shows the translation of $first$.

$$\begin{array}{c}
 \Gamma; loop \vdash e_1 \rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1}) \quad \Gamma; loop \vdash e_2 \rightarrow (q_{e_2}, \{item1\}, []) \\
 q \equiv (\Pi_{cols_{e_1}, iter1, pos1}(\sigma_{pos1 \leq cols_{e_1}.max+1}((q_{e_1}) \bowtie_{iter1=iter2} (\Pi_{cols_{e_1}.max+1:item1, iter2:iter1}(q_{e_2})))))) \\
 q \vdash itbls_{e_1} \xrightarrow{sel} itbls'_{e_1} \\
 \hline
 \Gamma; loop \vdash e_1.first(e_2) \rightarrow (q, cols_{e_1}, itbls'_{e_1}) \quad \text{(FIRST)}
 \end{array}$$

Box

In Section 4.3 the (un)boxing mechanism was introduced. The introduction of *box* expressions in the *AST* marks that we have to represent nested lists in the algebraic representation. The following rule describes how *box* expressions are handled in the translation process.

$$\frac{\Gamma; loop \vdash e \rightarrow qi_e \quad q \equiv @_{pos1:1}(\Pi_{iter1, item1:iter1}(loop))}{\Gamma; loop \vdash box(e) \rightarrow (q, \{item1\}, [item1 \rightarrow qi_e])} \quad \text{(Box)}$$

First we translate the expression e . The resulting queryinformationnode qi_e is now taken and written in the *itbls* component associated with the created column $item1$, that is put in the columnstructure as well. Now we need to construct a outer table with correct surrogate values. To do that we take the current loop context, preserve the $iter1$ column and take over the values of $iter1$ in the $item1$ column. In Figure 4.11 the boxing of a given table is illustrated.

4 Translation

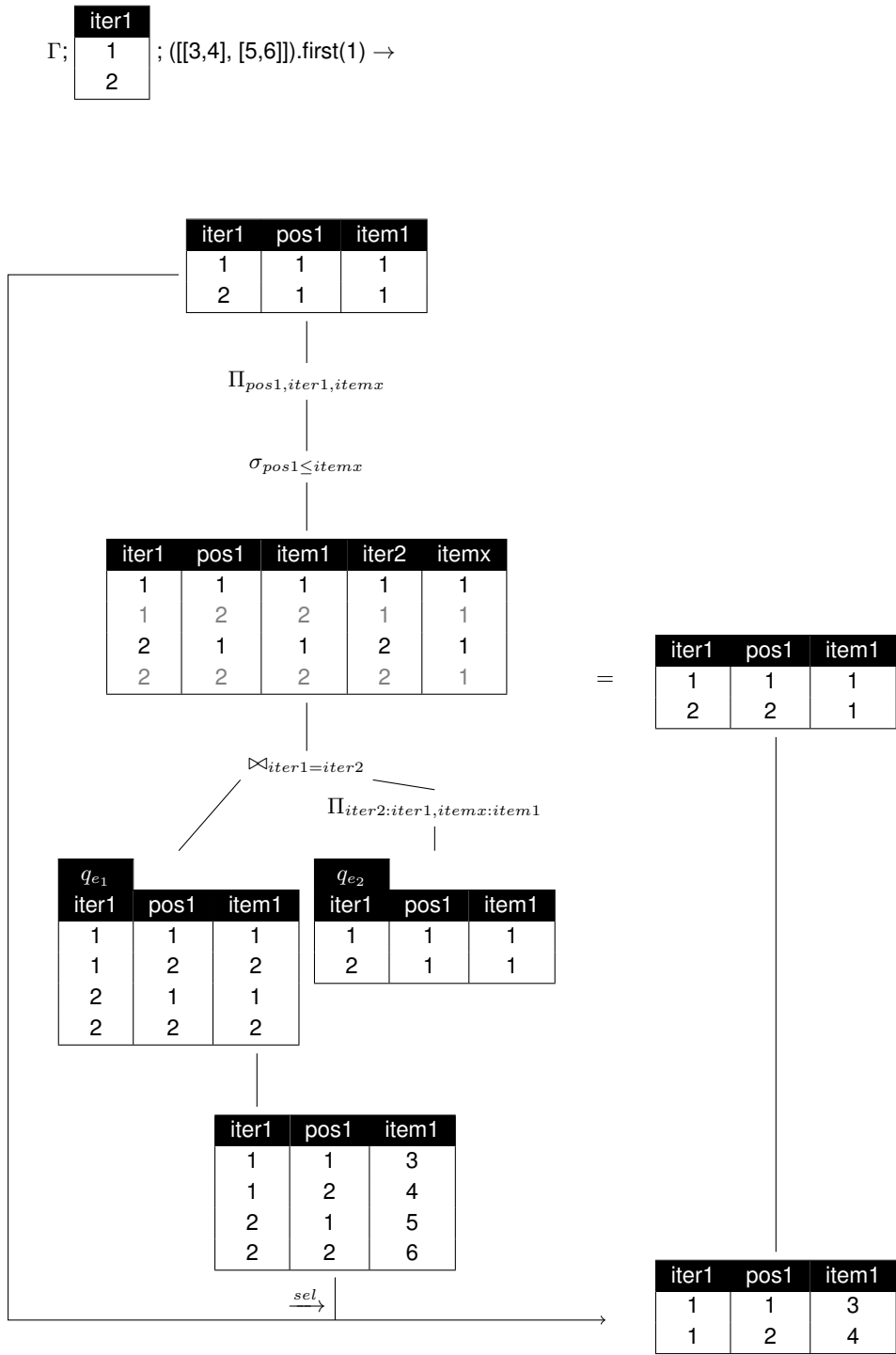


Figure 4.10: Example: First

4 Translation

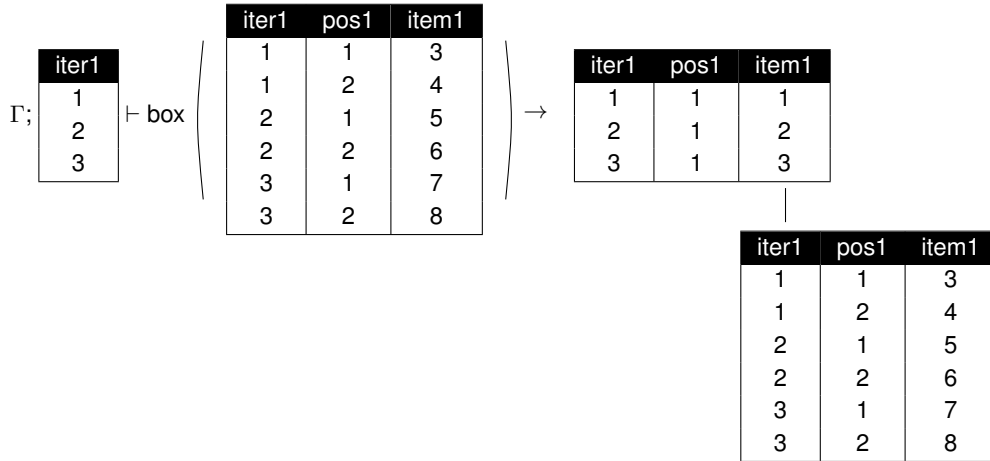


Figure 4.11: Example: Box

Unbox

In Section 4.3 the (un)boxing mechanism was introduced. The introduction of *unbox* expressions in the *AST* marks that we have to unnest nested lists in the algebraic representation. The following rule describes how *unbox* expressions are handled in the translation process.

$$\frac{\Gamma; loop \vdash e \rightarrow (qi_e, \{item1\}, [item1 \rightarrow qi])}{\Gamma; loop \vdash unbox(e) \rightarrow qi} \quad (\text{UNBOX})$$

First we translate the expression e . The resulting queryinformationnode contains qi_e that represents the outer table of the translated expression e . The column $item1$ is associated with the inner table qi in the surrogates of the queryinformationnode. Now we only need to extract qi out of the surrogates and discard the outer table. The overall result of the unbox operation is the extracted queryinformationnode qi .

Distinct

The compilation of the distinct operator is straight forward - at first e_1 is compiled which represents the list we have to apply the distinct operation on. Now we take the resulting plan q_{e_1} , project the $iter1$ column and every column contained in $cols_{e_1}$ to preserve them as well as their names, apply the distinct operator and finally attach a new $pos1$ column containing value 1 for every row. The resulting queryinformationnode contains the constructed plan q including the columns that were generated when compiling e_1 .

4 Translation

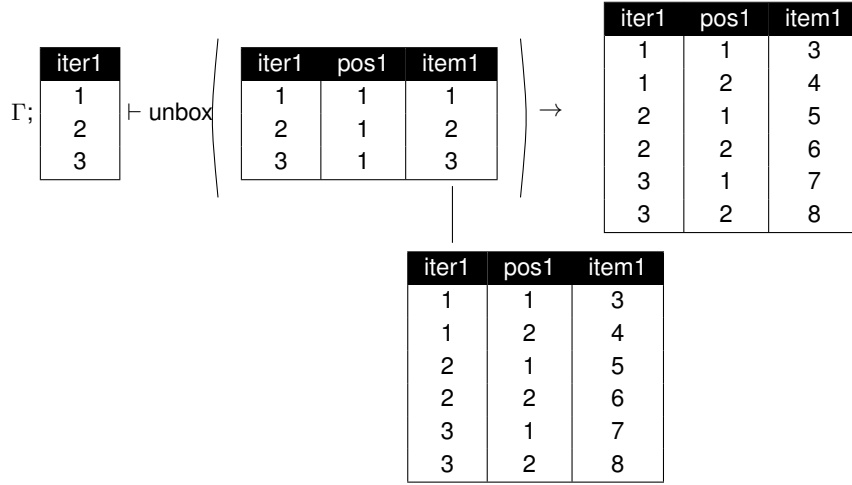


Figure 4.12: Example: Unbox

Note that we are neither able to restore the correct *pos1* column after the application of *distinct* nor can execute *distinct* on lists nested to arbitrary depth to date.

$$\frac{\Gamma; loop \vdash e_1 \rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1}) \quad q \equiv (@_{pos1:1}(\delta(\Pi_{iter1:iter1, cols_{e_1}:cols_{e_1}}(q_{e_1}))))}{\Gamma; loop \vdash e_1.distinct() \rightarrow (q, cols_{e_1}, [])} \text{ (DISTINCT)}$$

Union

To translate the union operator we first of all compile e_1 , the expression we invoke union on. Subsequently we compile e_2 serving as the expression to be merged with e_1 . The query plans q_{e_1} and q_{e_2} resulting from the compilation of e_1 and e_2 now get attached a new column *item0* which contains value 1 for q_{e_1} and 2 for q_{e_2} . The column *item0* is needed to preserve the order between the two lists after merging them to restore a correct *pos* column as well as creating new surrogate keys (*item'*). To calculate new surrogate keys the rownumber operation is applied after we have united q_{e_1} and q_{e_2} with the sorting criteria *iter1*, *item0* and *pos1*, all of them in ascending order. The resulting column *item'* now contains the new surrogate keys. Subsequently we restore the *pos* column of the relation by applying the rownumber operator again. The sort criterion remains the same as before to calculate the surrogate keys, but this time we use *iter1* as a partition column. The use of *iter1* as partition column enables us to calculate the correct *pos* values for every iteration, so the resulting column *pos2* contains the correct order of the

4 Translation

resulting relation. Finally we project out the columns not needed to restore the *iter*, *pos*, *item* form.

If there are any surrogates in the *itbls* components of the queryinformationnodes plan q containing the calculated new surrogate keys is passed to the function \xrightarrow{app} to handle them correctly (as prescribed in Section 4.1.2).

$$\begin{array}{c}
 \Gamma; loop \vdash e_1 \rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1}) \quad \Gamma; loop \vdash e_2 \rightarrow (q_{e_2}, cols_{e_2}, itbls_{e_2}) \\
 q \equiv \#_{item':(iter1:asc,item0:asc,pos1:asc)}(@_{item0:2}(q_{e_1}) \cup @_{item0:1}(q_{e_2})) \\
 q' \equiv \Pi_{iter1,pos1:pos2,cols_{e_1} \setminus keys(itbls_{e_1}),keys(itbls_{e_1}):item'(\#_{pos2:(iter1:asc,item0:asc,pos1:asc)}/iter1(q))} \\
 q \vdash (itbls_{e_1}, itbls_{e_2}) \xrightarrow{app} itbls' \\
 \hline
 \Gamma; loop \vdash e_1.union(e_2) \rightarrow (q', cols_{e_1}, itbls') \quad \text{(UNION)}
 \end{array}$$

Figure 4.13 shows exemplary how the union operator works.

The resulting queryinformationnode consists of the plan q' and a columnstructure holding the columns resulting from the compilation of e_1 . As already explained the union operator can only be invoked, if the column names as well as the column types are equal - it does therefore not matter if we use $cols_{e_1}$ or $cols_{e_2}$.

Intersect

The translation of intersect in fact is very similar to the translation of the union operator. First e_1 and e_2 are compiled, afterwards on the resulting plans q_{e_1} and q_{e_2} the intersect operator is applied and finally a new queryinformationnode is constructed, containing q and the columns of $cols_{e_1}$. Again, the chosen column structure does not matter as the columns' names and types also have to be equal when using the intersect operator.

$$\begin{array}{c}
 \Gamma; loop \vdash e_1 \rightarrow (q_{e_1}, cols_{e_1}, itbls_{e_1}) \\
 \Gamma; loop \vdash e_2 \rightarrow (q_{e_2}, cols_{e_2}, itbls_{e_2}) \quad q \equiv (q_{e_1} \cap q_{e_2}) \\
 \hline
 \Gamma; loop \vdash e_1.intersect(e_2) \rightarrow (q, cols_{e_1}, []) \quad \text{(INTERSECT)}
 \end{array}$$

Where

The *where* operator $e_1.where(\lambda x.e_2)$ is first of all translated to a expression similar to **for SEQ**($x \in e_1$) {**if** $\lambda x.e_2$ **then** x **else** SEQ}. So once again, we first have to

4 Translation

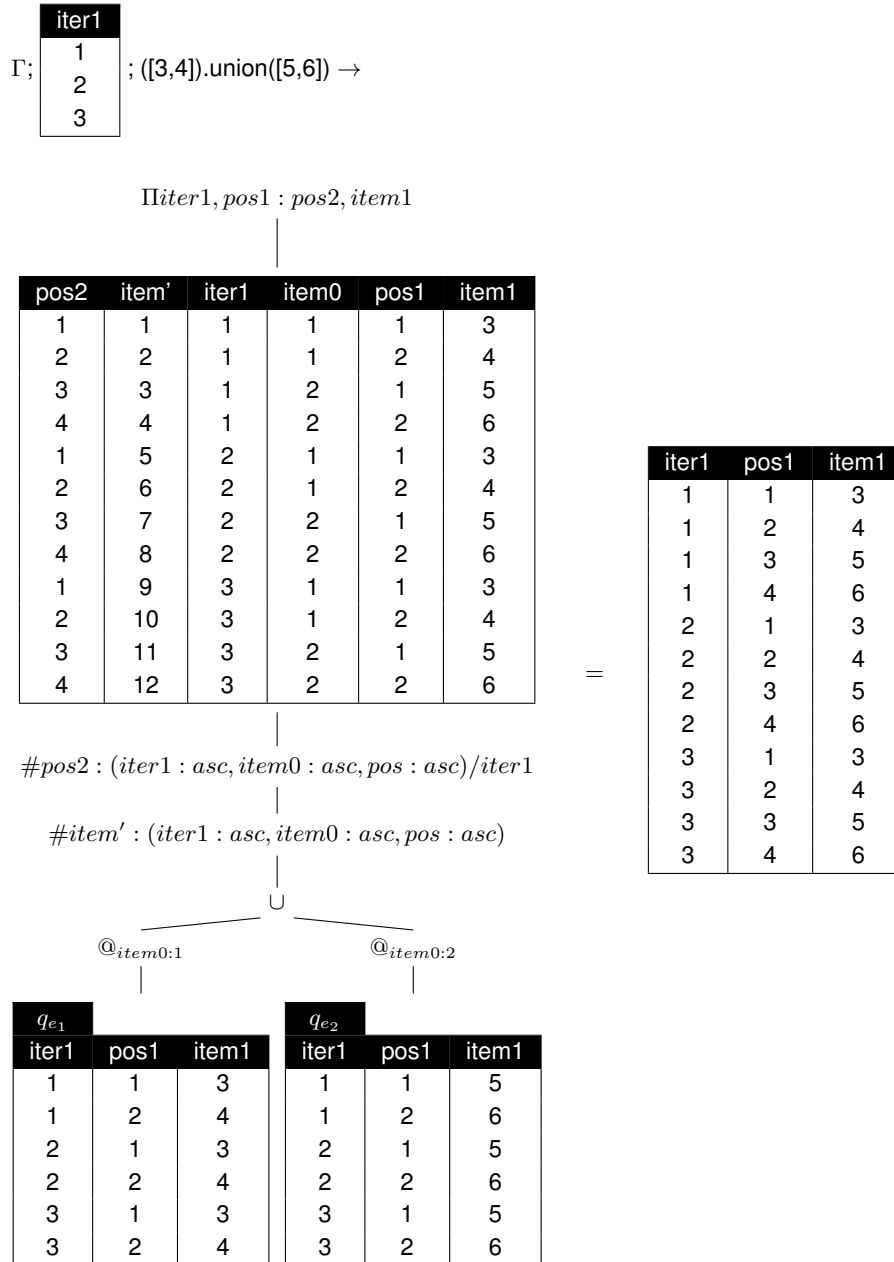


Figure 4.13: Example: Union

translate the generated **for** expression. That means that we already have query-informationnodes bound to the environment (in ① denoted as x), so in the context of the **for** expression the **if then else** expression is translated. In ① we compile the function that serves as condition of the **for** statement. This translation will deliver a queryinformationnode containing plan q_{e_2} , a single column $item1$ holding the boolean results of the function invocation and no surrogates. Subsequently we select the rows from q_{e_2} that were evaluated to true by the function invocation by

4 Translation

applying the select operator (②). As we only need the *iter1* and *pos1* column of the selected rows, we apply the project operator and rename the column *iter1* to *iter2* and *pos1* to *pos2* to prevent name clashes in the following join. In step ③ we take q_x from the environment as it contains the list we iterate over - so it holds the proper rows we need to combine with the function results. To combine q_x and res_{then} we join the two plans on their *iter* and *pos* columns and finally preserve the columns *iter1*, *pos1* as well as every *item* contained in the $cols_x$ component. The result now contains the rows that satisfied the function $\lambda x.e_2$ only - but with their proper values. In step ④ a empty list is constructed and contains the columns *iter1*, *pos1* and $cols_x$ as well as we need to unite the empty table with $q_{x_{then}}$ in step ⑤. The empty list is always used here, as a *where* operation never uses a **else** condition. In step ⑤ we finally unite the empty list with $q_{x_{then}}$, restore the correct order by using the rownum operation with order *iter1* : *asc*, *pos1* : *asc* partitioned by *iter1* in the *pos2* column and apply the project operator to rename *pos2* to *pos1* and preserve the *iter* column as well as the columns of $cols_x$. As last step we need to adapt possible inner tables by using the \xrightarrow{sel} function (⑥).

In Figure 4.14 an example for the *where* operation is shown.

$$\begin{array}{l}
 \textcircled{1} \quad [\dots, x \rightarrow (q_x, cols_x, itbls_x), \dots]; loop \vdash \lambda x.e_2 \rightarrow (q_{e_2}, \{item1\}, []) \\
 \textcircled{2} \quad res_{then} \equiv (\Pi_{iter2:iter1, pos2:pos1}(\sigma_{item1=true}(q_{e_2}))) \\
 \textcircled{3} \quad q_{x_{then}} \equiv (\Pi_{iter1, pos1, cols_x}(res_{then} \bowtie_{iter2=iter1, pos2=pos1}(q_x))) \\
 \textcircled{4} \quad q_{else} \equiv \left(\begin{array}{|c|} \hline \mathbf{a} \\ \hline \end{array} \right) \\
 \textcircled{5} \quad q \equiv \Pi_{iter1, pos1:pos2, cols_x}(\#_{pos2:(iter1, pos1)/iter1}(q_{x_{then}} \cup q_{else})) \\
 \textcircled{6} \quad q \vdash itbls_x \xrightarrow{sel} itbls' \\
 \hline
 [\dots, x \rightarrow (q_x, cols_x, itbls_x), \dots]; loop \vdash e_1.where(\lambda x.e_2) \rightarrow (q, cols_x, itbls') \quad \text{(WHERE)}
 \end{array}$$

Constant

A *Constant-expression* gets translated to its *loop-lifted* representation.

$$\frac{}{\Gamma; loop \vdash v \rightarrow ((@_{item1:v}(@_{pos1:1}(loop)), \{item1\}, [])} \quad \text{(CONSTANT)}$$

So we take the given *loop-context*, attach the column *pos1* with value 1 to every row and finally attach *item1* with the value specified by v to every row as well. As a

4 Translation

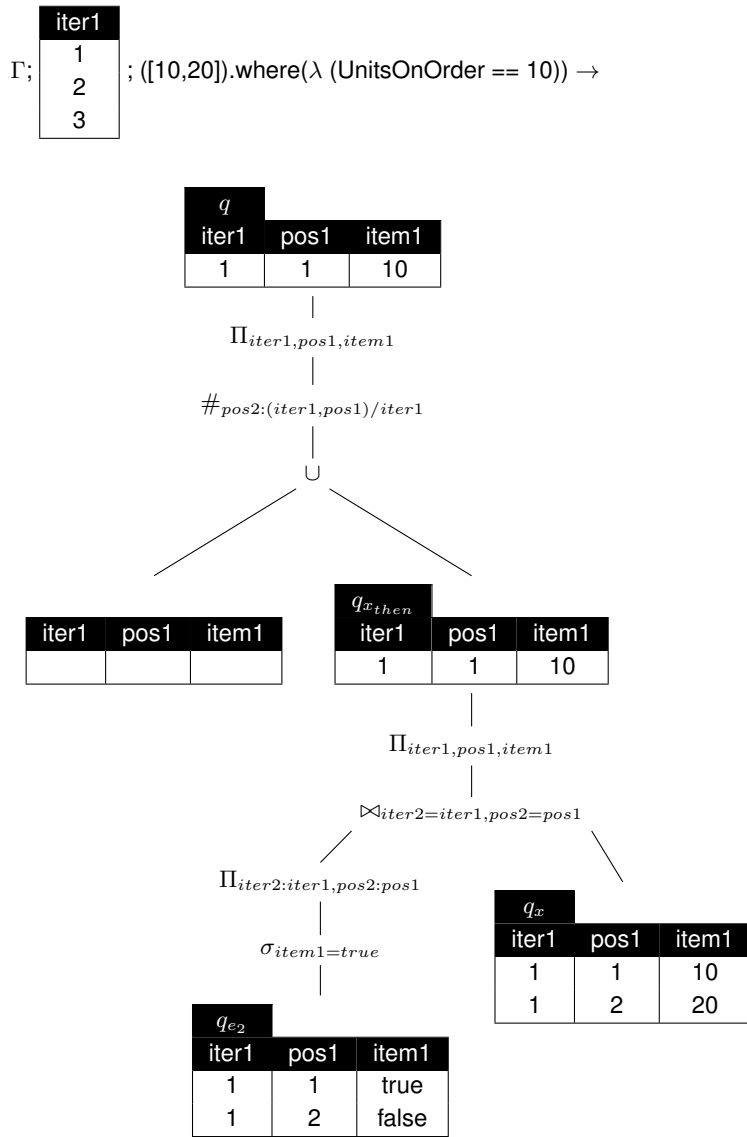


Figure 4.14: Example: Where

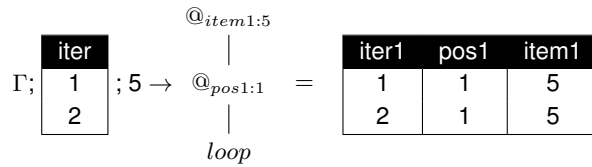


Figure 4.15: Example: Translation of a constant

consequence, we have to insert *item1* to the column structure and leave the list of surrogates empty because the *Constant*-rule does not construct any inner tables.

4 Translation

Figure 4.15 shows the translation of a constant 5 with the given *loop* context containing two rows.

Binary and Unary Operations

A $\odot()$ operation marks the invocation of a binary or unary operation. So the $\odot()$ expression holds the **operator** to be used as well as the **arguments** to operate on.

$$\begin{array}{c}
 \odot \in \{+, -, \times, \backslash, \text{mod}, \vee, \wedge, =, >\} \\
 \Gamma; \text{loop} \vdash e_1 \rightarrow (q_{e_1}, \{\text{item1}\}, []) \quad \Gamma; \text{loop} \vdash e_2 \rightarrow (q_{e_2}, \{\text{item1}\}, []) \\
 q \equiv ((\Pi_{\text{iter2}:\text{iter1}, \text{item1}+1:\text{item1}}(q_{e_1})) \bowtie_{\text{iter2}=\text{iter1}} (q_{e_2})) \\
 q' \equiv \Pi_{\text{iter1}, \text{pos1}, \text{item1}:\text{item1}+2}(\odot_{\text{item1}+2:(\text{item1}+1, \text{item1})}(q)) \\
 \hline
 \Gamma; \text{loop} \vdash \odot(e_1, e_2) \rightarrow (q', \{\text{item1}\}, []) \quad \text{(BINARY OPERATIONS)}
 \end{array}$$

So first of all we compile the expressions e_1 and e_2 which we expect to be of *Row* type for each iteration. In order to be able to use the \odot operator, we apply the project operator on the generated plan of e_1 , rename the columns *iter1* and *item1* to *iter2* and *item1+1* and use an equi-join to combine the plan of e_1 and e_2 . On the resulting plan q we now apply the binary operation and rename the result column by using the projection operator again. Figure 4.16 illustrates the translation of binary operations exemplary using the \times operator.

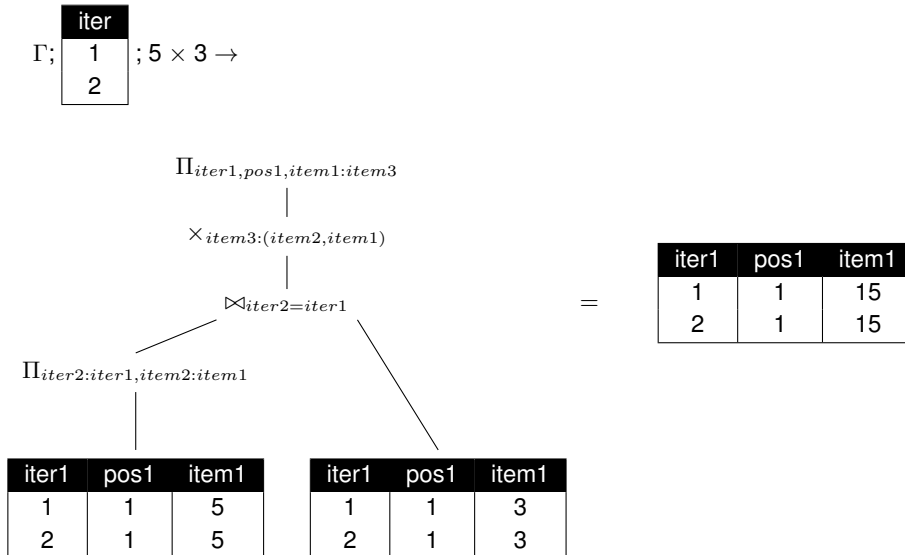


Figure 4.16: Example: Translation of binary operations

4 Translation

Target operation	Representation
$e_1 < e_2$	$e_2 > e_1$
$e_1 \leq e_2$	$!(e_1 > e_2)$
$e_1 \geq e_2$	$!(e_2 > e_1)$
$e_1 \neq e_2$	$!(e_1 = e_2)$

Table 4.1: Representation of missing comparison operations

Because PATHFINDER does support a complete base of comparison operators only, namely $>$, $=$, \neg , we have to derive the operations $<$, \leq , \geq and \neq . Table 4.1 shows how the missing operations are represented.

So the above rule is applicable for $<$ operation as well, but we have to exchange the positions of the operands and use $>$. For the operations \leq , \geq and \neq we use an own rule, where we have to exchange operators for the \geq operation!

$$\begin{array}{c}
 \odot \in \{\leq, \neq\} \\
 \Gamma; loop \vdash e_1 \rightarrow (q_{e_1}, \{item1\}, []) \quad \Gamma; loop \vdash e_2 \rightarrow (q_{e_2}, \{item1\}, []) \\
 q \equiv ((\Pi_{iter2:iter1, item1+1:item1}(q_{e_1})) \bowtie_{iter2=iter1}(q_{e_2})) \\
 q' \equiv \Pi_{iter1:iter1, pos1:pos1, item1:item1+3}(\neg_{item1+3:(item1+2)}(\odot_{item1+2:(item1+1, item1)}(q))) \\
 \hline
 \Gamma; loop \vdash \odot(e_1, e_2) \rightarrow (q', \{item1\}, []) \quad \text{(COMPARISON)}
 \end{array}$$

The unary operation \neg is translated as follows:

$$\frac{\Gamma; loop \vdash operand \rightarrow (q_{operand}, \{item1\}, []) \quad q \equiv \neg_{item1+1:(item1)}(q_{operand})}{\Gamma; loop \vdash COMPLETE \rightarrow (q, \{item1 + 1\}, [])} \quad \text{(NOT)}$$

Var

A *Var*-expression holds a **name** that is used in the translation rule to retrieve and return a queryinformationnode qi that is bound to the given **name** in the environment.

$$\frac{}{[\dots, name \rightarrow qi, \dots]; loop \vdash name \rightarrow qi} \quad \text{(VAR)}$$

4 Translation

TableDot

A *TableDot*-expression represents a table to be referenced existing in the underlying database. We translate the expression with respect to the given key that defines the order, the given columns and the current *loop* context to its loop lifted representation.

$$\frac{\text{key} = \{k \in \text{cols}\} \quad \text{cols} = \{c_1, \dots, c_n\} \quad q \equiv ((\text{rnk}_{\text{pos1}:(\text{key}:\text{asc})}(\text{⊕}_{R(\text{cols})})) \times \text{loop})}{\Gamma; \text{loop} \vdash \text{TableDot}R(\text{cols}) \rightarrow (q, \text{cols}, [])} \quad (\text{TABLEDOT})$$

The example in Figure 4.17 shows the access on the database table *Products* with columns *id*, *name* and *price* where *id* represents the primary key to sort by. The given *loop* context contains two rows that serve to unfold the stored database rows to multiple iterations.

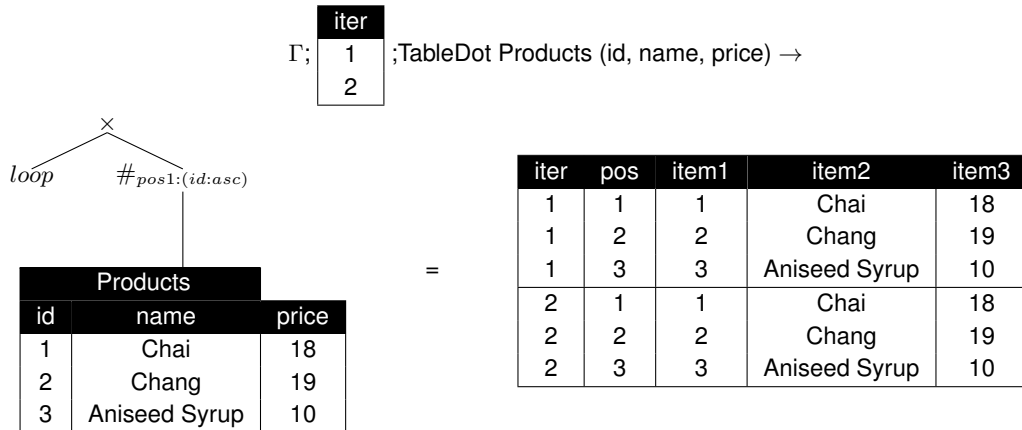
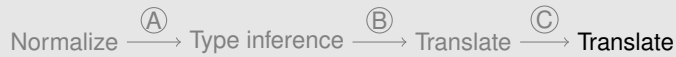


Figure 4.17: Example: ⊕ operator



4.5 Translate to SQL

The next step in the translation process is to construct a XML¹ representation of the query plans generated in the translation. A query plan received from the translation consists of a tree of queryinformationnodes that represent the translated BATCHES program. To be able to serialize the constructed query plans to XML, we need a representation that describes the structure properly. In Figure 4.18 a rough overview over the XML representation is given.

$$\begin{aligned}
 \text{query_plan_bundle} & ::= \overline{x \rightarrow \text{query_plan}} \\
 \text{query_plan} & ::= \text{logical_query_plan} \\
 \text{logical_query_plan} & ::= \overline{\text{node}} \\
 x & \in \mathbb{N}_0
 \end{aligned}$$

Figure 4.18: Structure of query plan bundles

A *query_plan_bundle* represents the root element of the XML document. It contains an arbitrary count of *query_plan* nodes that are associated with a number $x \in \mathbb{N}_0$. The number later on serves to identify what plan belongs to which queryinformationnode. A *query_plan* contains a single *logical_query_plan* that on his part holds an arbitrary number of *nodes*. Nodes finally represent single algebraic operators as constructed in the translation phase.

Now we need to clarify, how our queryinformationnode structure is mapped to a *query_plan_bundle*. Every queryinformationnode has a associated *query_plan* in the *query_plan_bundle* that contains the plan stored in the queryinformationnode.

Plans of that form are now given to the PATHFINDER compiler. PATHFINDER compiles and optimizes the passed *query_plan_bundle* and constructs a query for every single *query_plan* contained in the *query_plan_bundle*. What PATHFINDER returns is again an XML file of the structure as illustrated in Figure 4.19.

A *query_plan_bundle* again contains an arbitrary count of *query_plan*. The *query_plan* nodes are associated with the same number x as they were in the input file. A *query_plan* contains *properties* that store informations about plans. We use it to store the information, if the plan of the queryinformationnode is expected to be sin-

¹Extensible Markup Language

4 Translation

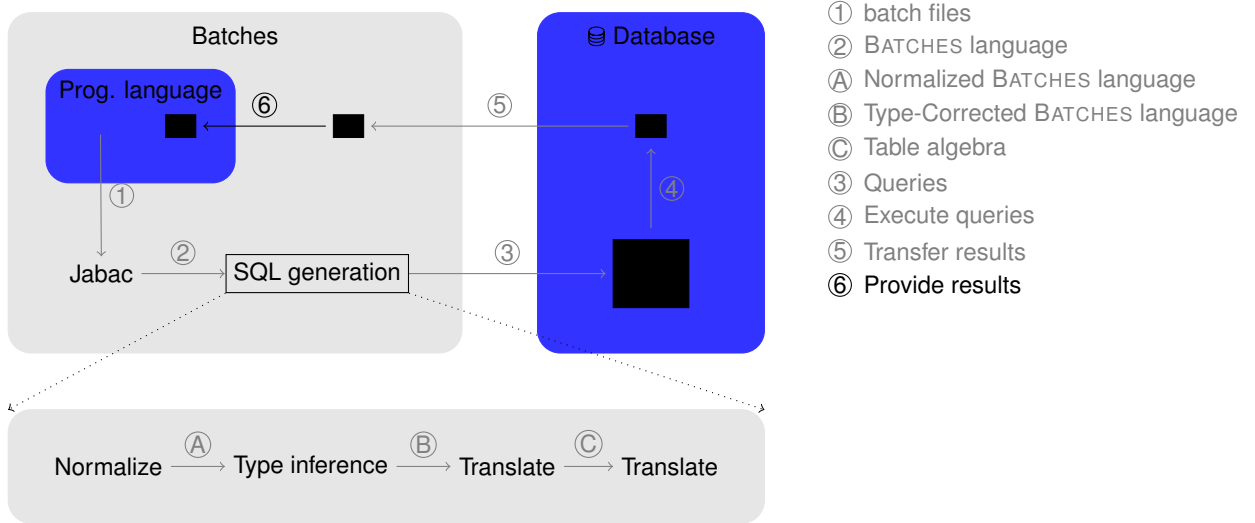
<i>query_plan_bundle</i>	::=	$\overline{x \rightarrow query_plan}$
<i>query_plan</i>	::=	<i>properties, schema, query</i>
<i>schema</i>	::=	\overline{column}
<i>query</i>	::=	string
$x \in \mathbb{N}_0$		

Figure 4.19: Structure of PATHFINDERS' result

gle or multi valued. Furthermore a *query_plan* contains a *schema* that on his part contains the columns that result from the *query* with their name, their function (iter, pos or item) and their position. The *query* node contains a SQL query string.

So now we are able to execute the queries on the database.

4 Translation



4.6 Providing results

As a final step in translation process we need to provide the results received from the queries to the *Forest* data structure. As a reminder the structure of a *Forest* as defined in Section 2.4.2 is shown again in Figure 4.20.

$F \in Forest$	$::= MV MAF F \overline{M}$
$MF \in MultiForest$	$::= LF F$
$MV \in Values Map$	$::= [I \rightarrow T O]$
$MAF \in MultiForest Map$	$::= [I \rightarrow MF]$
$LF \in Forest List$	$::= [F]$
$M \in Method$	$::= T m(\overline{T p});$
$T \in Type$	$::= P I$
$P \in Primitive$	$::= Void Number Boolean String$ $ A Duration RawData$
$A \in Date$	$::= Date DateTime$
$N \in Number$	$::= Integer Float Decimal$
$I, m, p \in Name$	
$O \in Object$	

Figure 4.20: Definition of the *Forest* data structure

The datastructures of the translation involved in the filling of the *Forest* are:

1. *Tree of query information nodes*

The tree of query information nodes contains the informations how to fill the *Forest*. Query information nodes provide the information, if we have to merge

4 Translation

results by holding the surrogates as well as to which plan in the *query_plan_bundle* they belong. Furthermore they contain the outputs with their names to be bound.

2. *query_plan_bundle*

The *query_plan_bundles* provide us with the information what the schema of the plan is. This information is used to read out the resultset delivered by executed queries. Additionally each *query_plan* contains the information if the result we expect has to be multi or single valued.

So to fill the *Forest*, we first of all take the topmost queryinformationnode and check its result is expected to be single or multi valued. If it is single valued we know that we don't have to create a new *Multiforest* but can write results to the current *Forest*. If the result is expected to be multi valued, we have to create a new *Multiforest* and for every row a *Forest* has to be attached to the new *Multiforest*. To construct a *Multiforest* we need to know the variable the results are bound to - to determine that information the outputs component of the queryinformationnode is used where the keys represent the proper variables. Values put into a *Forest* are bound to specific names - these names are stored in the output component of the queryinformationnode as well. To write the results from the resultset to the *Forest* we now need the information where we can find the values to be written. That means that we have to know if the values can be found in the resultset of the current queryinformationnodes' plan or we have to write values from a resultset that was generated by a query that is referenced in the *itbls* component of the queryinformationnode. To determine that information, we take the column we want to read out and make a lookup in the *itbls* component if the column is present there. If it is present we know that we have to fill in the values of the resultset of a referenced plan. If we determine that the column actually references a inner list, we have to fill in the values of the resultset of the inner list by taking a result of the column of the outer resultset and compare it to the iter column of the inner resultset to combine the values correctly.

As the columns are aligned in the resultset with the positions that are provided in the *query_plans schema* component, we are able to take rely on the positions provided in the *schema* to read out the correct columns.

So a tree of queryinformationnodes is processed as follows:

1. Take a queryinformationnode
2. Determine if it is single or multi valued
3. Read out the columns of the queryinformationnode from *query_plan_bundle*

4 Translation

and use that information to read the proper values from the resultset. If a surrogate column occurs handle it correctly by reading results from the correct resultset and combining it with the correct item column of the outer table.

4. Take the next queryinformationnode from the *itbls* component and start over with step one.

5 Conclusion

As the gap between programming languages and databases has been a subject in research for many years now with many different approaches with its specific strengths and weaknesses BATCHES is a new elegant way of combining the two shores. Actually BATCHES has its weaknesses when it comes to executing queries on different database backends as well as the set of operations properly working. By embedding the FERRY compilation techniques we were able to abolish the problem of supporting different database backends as well as to augment the operator toolkit. Knowing that the FERRY integration actually is a good step for BATCHES in the right direction. In future work the operator toolkit could be further expanded, as BATCHES' architecture supports the easy integration of new operators. In fact the BATCHES approach has the potential to simplify the life of many programmers.

Bibliography

- [1] Eli Tilevich William Ali Ibrahim, Yang Jiao. Remote batch invocation for compositional object services.
- [2] Marc Fisher II William R. Cook Eli Tilevich Ali Ibrahim, Yang Jiao. Remote batch invocation for web services. 2009.
- [3] William R. Cook. Batch service manifesto and user manual.
- [4] Ezra Cooper. The script-writer's dream: How to write great sql in your own language and be sure it will succeed.
- [5] Yang Jiao Eli Tilevich, William R. Cook. Explicit batching for distributed objects.
- [6] Manuel Mayr. A sql:99 codegenerator for pathfinder. Master's thesis, 2007.
- [7] Tom Schreiber. Translation of list comprehensions for relational database systems. Master's thesis, 2008.
- [8] Jan Rittinger Tom Schreiber Torsten Grust, Manuel Mayr. Ferry - database-supported program execution. 2009.
- [9] Jens Teubner Torsten Grust. Relational algebra: Mother tongue-xquery: Fluent. page 8, 2004.
- [10] Alexander Ulrich. A ferry-based query backend for the link programming language. Master's thesis, 2010.
- [11] Ali H. Ibrahim William R. Cook. Integrating programming languages & databases: What's the problem?

Name: Dennis Butterstein

Matrikelnummer: 3307933

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine Anderen, als die angegebenen Quellen und Hilfsmittel verwendet habe. Der Inhalt der vorliegenden Arbeit wurde von mir noch nicht als Prüfungsleistung eingereicht.

Tübingen, den

Dennis Butterstein