UNIVERSITA' DEGLI STUDI MILANO

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

CORSO DI LAUREA IN TECNOLOGIE DELL'INFORMAZIONE E DELLA COMUNICAZIONE

The Construction of a Type-Directed LINQ to SQL Provider

Relatore: Prof. Marco MESITI
Correlatore: Prof. Torsten GRUST

Tesi di laurea di:
Simone BONETTI
Matr. n. 736467

Anno Accademico 2008/2009

I would like to dedicate this thesis to my parents Lorenzo and Sandra, my sister Silvia, my relatives and all the friends that has supported me over these years, letting me taking my decisions without any restriction. Especially I want to dedicate it to my grandfathers that cannot be here to witness it.

Acknowledgements

Me, myself and I in all the versions on the parallel (or even perpendicular or *moved on*) universes, whose are thanking the respective parallel (or perpendicular or *moved on*) versions of themself.

Seriously..

There are two persons, without whom probably I wouldn't be where and who I am now, that deserve a very special thanx:

My big brother Davide (and his wife Muriel) for many years of constant growing friendship, for all the great experiences we have shared, for the sleepless talking night, for always been there to push and support me and for always making me open my eyes and see things in the right way. Without you probably I would have stopped much earlier! Thanx for believing in me and force me to do the same! I'll rise a Saraceno in your honor!

Katja, und natürlich auch Sarah und Gordon, für die Freundschaft – die großartige Zeit, die wir zusammenverbracht haben. Dir, Katja, danke ich besonders für die (von dir verhassten) nächtlichen Gespräche und dafür, dass du in dieser Zeit meine Konstante warst, wenn alles schief zu laufen schien. Als Mensch habe ich mich sehr weiterentwickelt und ich danke dir für die gute (und manchmal auch schlechte) Zeit. Die Zeit mit dir war etwas Besonderes, und ich werde Sie für immer in Erinnerung behalten. Ich hoffe, dir geht es genauso, wenn du daran denkst. Unsere Beziehung wird weiterwachsen und noch sehr lange bestehen bleiben. Ich danke dir, dass du es geschafft hast, an mich zu glauben und mir zu verstehen gegeben hast, dass ich im Grunde gar nicht so schlecht bin. Von ganzem Herzen, meine Pappnase!!! ... schwarz, wie üblich!!

Many people are in a way or another important for me, especially the following:

Mattia *'secchiello'* Berera, although is not true anymore, for the many years of true friendship and funny experiences. Hope you'll find again the stoppato way! Polyphonic finger-style and jazz it's not for you! Only straight palm muted E is the right way!

Tom Schreiber and his Ferry project, which is the coolest thing ever!! Thanx for all the patience, help and lessons over those months.

Torsten Grust, Manuel Mayr, Jan Rittinger, Melanie Herschel and Monika Weber for being so welcoming with me, and offer me this great working and human experience. Thanx for giving me all the necessary resources and all the possible help to work on this great argument.

(Keller) Bier und Brezel, ohne sie würde diese Thesis nicht entwickelt worden.

Chaos, order and complexity, for constantly challenging our intellectuality.

Keeper: again Davide, again Mattia and Press for the great and way too powerful years of musical experience. Stacchettosa still rules!

ALAN: Chicco *'this head is different, it has one knob more than the old one'*, Mosso *'Holz vor der Hütte'*, Spini and Gian for the even short but good times in the smelly rehearse room.

Sara, for the interesting talks over book, general culture and for suggest me new things: it's good to see that in the world there are still persons that can really talk about more than one single argument. And thanx for being sad if I'll go away.

Hyun-Keun, Tobias und Lilian, die beste WG der Welt, für die belustigende Zeit in Tübingen!!! HK, die Skalpelle werden in der richtigen Weise benutzt werden, ich verspreche dir!

Paul Lambs, for the evening talks and the neverending research of new musical shores! There is more than one of everything, remember! EDIT: Thanx also for the thanx enhancement!

Sara again, even if she didn't send me any letter while I was away, and Elena, which should eat more animal proteins, for the little book club. I'll finish DC as soon as possible! I swear!!!!

Storchen, Kuckuck, Clubhaus und die anderen coolen Kneipen in Tü!

The Irish people: Poncia, Franco and Tatiana, for the time before and for the holidays up there!

The Pink Side of the Force, for always come out in the wrong moments (usually around 4 AM)! Thanx HK and MM, never stop looking at the starts!!

Gianluca, for introducing me to the computer science world when I was a kid.

Monica, for the totally senseless, but in a strange way not so much, talking. May our path always cross at same point in our life.

Silvia, please, don't destroy the cars because someone still needs them!

Sebastiano, good luck with the search of new band members!

Paolo *'bardo'* Manzi and all the staff of holymetal.com.

The people from the University of Milano:

My project-mate Andrea *'when MM shows up, I'll eat his heart'* Bozza for all those years of collaboration, friendship and craziness at the university. The Matrix String one day will rule the world!!

Ilario for sharing opinions on what's really important during studies: the Dark Tower and good quality beers.

Laura Cirocco, for the cooooooooool diagrams on the FSD project. Actually..for the sweet cakes!!

Massimo Matozzo...there are 12 potatoes hidden in your flat! Find them!!

Nicola Palmieri..maybe I'll visit your website!..one day!
Marco Alari, for being such a reliable and remarkable test-user, as seen in 3.3 and 4.3! BTW, Dana dies at the end of season 3, and Locke IS dead!
Laura, because she forced me to not change the incipit of the acknowledgment list and because she told me about the Masaia land.
All the people I've cross path with during the time there.

Christian, Simone, Gian Pietro, Roby and Guttalax for the funny years and the genius jokes in the flat in Milano.
Holly *'lab53'*, Roby *'Mr Ripetute'*, Keelyn *'cool, it doesn't have any gas in it'*, Sandro *'Waldgeist'* and the other friends here.
Music, that always had an important role on my life, through good and bad times.
Lost, for the chapter's coolest titles!
Doc Brown for the PC!
Giuseppe Rossi, Aldo, Martin and all the other people in Intersect/Syrea.
All the friends for some reasons lost during these years: along the path you were probably important, so you deserve a mention.

Probably someone is still missing, and I'm deeply sorry for that.

Contents

1 Introduction **1**

2 Software overview **5**

2.1 LINQ 5

2.1.1 The iEnumerable interface 6

2.1.2 The iQueryable interface 7

2.2 The TPC-H Database 9

2.3 Pathfinder 9

3 Normalization of the linq tree **11**

3.1 Expression trees 12

3.2 The new light-weighted AST 14

3.2.1 The AST super class 16

3.2.2 Binary Operators 17

3.2.3 Function Call 17

3.2.4 Constant 17

3.2.5 Lambda Expressions 18

3.2.6 Member Access 18

3.2.7 Not Implemented 18

3.2.8 Record 19

3.2.9 Select 19

3.2.10 Table 19

3.2.11 Unary Operator 20

3.2.12 Variable 20

3.2.13 Walkabout 21

3.3 Too many SQO's overloads 22
3.3.1 Newly introduced operators 23
3.3.2 Aggregates 23
3.3.3 All 24
3.3.4 Any 25
3.3.5 Concat 25
3.3.6 Count 26
3.3.7 Distinct 26
3.3.8 ElementAt 27
3.3.9 First 27
3.3.10 GroupBy 28
3.3.11 Select 29
3.3.12 SelectMany 30
3.3.13 Single 30
3.3.14 Skip 31
3.3.15 Take 31
3.3.16 Where 31
3.4 Go back to 1 Normal Form 32
3.4.1 Un-virtualizing 33
3.4.2 The TableMap environment 33
3.4.3 The Mapping Rules 35
3.4.4 The Join branch 41
3.5 (Un)Boxing the tree 45
3.5.1 Introduction rules 47
3.6 The Running Example 52
3.6.1 The LINQ Query 52
3.6.2 The LINQ expression tree 52
3.6.3 The SQO-Normalized tree 52
3.6.4 The Un-virtualized tree 52
3.6.5 The (Un)Boxed tree 55

4 Algebraic compilation **59**
4.1 The Pathfinder Algebra . . . 59
4.1.1 Binary Operators . . . 62
4.1.2 Unary Operators . . . 65
4.1.3 Leaf Nodes . . . 69
4.2 Types mappings . . . 70
4.3 Loop-lifting . . . 72
4.4 Compilation Rules . . . 72
4.4.1 The *QCsTs* structure . . . 73
4.4.2 Helper functions and operators . . . 75
4.4.3 Aggregate . . . 78
4.4.4 Binary . . . 79
4.4.5 Box . . . 79
4.4.6 Concat . . . 80
4.4.7 Conditional . . . 80
4.4.8 Constant . . . 81
4.4.9 Count . . . 82
4.4.10 Distinct . . . 82
4.4.11 ElementAt . . . 83
4.4.12 Filter . . . 83
4.4.13 Flatten . . . 84
4.4.14 GROUP . . . 84
4.4.15 Member Access . . . 85
4.4.16 Not . . . 86
4.4.17 Record . . . 86
4.4.18 Select . . . 87
4.4.19 Single . . . 89
4.4.20 Skip . . . 89
4.4.21 Table . . . 89
4.4.22 Take . . . 90
4.4.23 UnBox . . . 90
4.4.24 Variable . . . 91
4.5 The Running Example . . . 92

4.5.1 The algebraic plan 92

5 We have to go back 95

5.1 Xml representation 95

5.2 PathFinder 95

5.3 Heap results 96

5.4 The Running Example 97

5.4.1 The optimized plan 97

5.4.2 The generated SQL code 97

6 Replacing the Leader 105

6.1 Creating the provider 105

6.1.1 Overview of the interfaces 105

6.1.2 Implementation of the skeleton of the provider 107

6.2 Generating the Data Context 108

7 Conclusions 111

Bibliography 115

Chapter 1

Introduction

"'Guys? Where are we?"'

1. INTRODUCTION

The LINQ technology, an extension for the .NET framework, extends the primary language with a set of query operators that can be used to query theoretically any data source [6]. It introduces a new level of abstraction between the code and the data that handles the compilation and execution of the given query, allowing the user to use the same syntax and operators to write queries against different data sources. Inside this tier the query will be translated by a data source specific *provider* into the right Domain Specific Query Language and executed against the data source.

The Microsoft LINQ-to-SQL provider, which handles LINQ queries against databases, uses an approach which is *data driven*, meaning that the number of generated queries is proportional to the queried data, leading to the generation of maybe million of queries. In case of complex query on huge amounts of data the query will need a long execution time or it will even not terminate due to heap failure.

This happens because of how query comprehensions are handled. Query comprehension is the principal LINQ construct and has that form:

$$e_1.Select(v \;\Rightarrow\; e_2)$$

At each iteration of e_1, v is bound to a new value and then e_2 is evaluated based on this binding. So there will be n (where n represents the cardinality of e_1) independent evaluation of e_2 under different bindings of v.

In this thesis is presented a new running LINQ-to-SQL provider, which is *type directed*, meaning that the number of generated SQL queries will depend only on the object structure of the result [3, 9].

The query will be compiled into an intermediate algebra in order to use the facilities offered by the *Pathfinder* [1] engine to optimize [7] it and generate the SQL code [2]. What is different from the actual Microsoft translation approach is that query comprehensions are handled in a different way, by moving their evaluation inside the DBMS, which is optimized to work with huge amount of data.

One of the main features of relational query processors is to work on set oriented evaluation: in absence of inter-row dependencies, the system may process them in any order or even in parallel. Using an adapted version of the *loop lifting* compilation technique [5, 4], that compiles

a comprehension into a single table with all bindings for all iterations, this independence is realized. The full potential of a relational query processor can then be used to evaluate the query.

This produces only one SQL query that evaluates all the iterations of a query comprehension at once. The *loop lifting* approach maintains track of the independent iterations and each variables scope in order to handle also nested query comprehensions in the same way, having at end only one SQL query that evaluates the whole LINQ query. This lead the amount of query to be independent from the data size. Data size will just affect the execution time.

This compilation technique adds two new columns to the *loop lifted* generated table: an *iter* column, that keeps track of the individual iterations, and a *pos* column, that represents the position of a tuple inside an iteration. This will allow the new provider to support also SQOs that adhere to proper order semantics, like the ElementAt operator, not supported by the actual LINQ-to-SQL provider.

The provider is *type directed* because a SQL query will be produced for each list constructor appearing in the result to represent its values. The LINQ technology allows the user to shape the result's type as he wants, meaning it can have complex structures such as nested lists inside it. The relational back-end works only on flat types in first normal form, therefore those nested data structures must be simplified. Using *surrogate values* [8], all nested lists inside the relational representation of the result will be substituted by surrogate values and a new single table containing the relational encoding of all the inner lists will be generated. So each level of nesting will be represented by a relational table (described by a SQL query) linked by surrogate values. That also allows the result to be consumed gradually at different nesting levels, based on what the end user will access: a query representing a nesting level will be then executed on the back end only when data inside it is accessed.

The result's plan will be formed by at least one SQL query representing the most outer query and by a SQL query for each nesting level in the result. The number of generated query is therefore totally independent from the queried data size.

Another advantage of this approach is that the generate SQL code, contrary to the Microsoft provider, can be executed on any SQL:1999 capable DBMS because it doesn't depend, like the Microsoft provider, on Microsoft SQL Server's behavior.

The end result will have a little overload due to the normalization process required but it is a cost well worth paying given the resulting advantages.

In chapter 3 the LINQ expression tree generated from the LINQ query is normalized in order to be compiled. As first step the tree is pruned of the unnecessary SQOs in order to have less compilation rules at compile time.
In LINQ the database table are mapped into the objects. That allows the user to navigate through them using the foreign keys creating *virtual* attributes. To be executed on a pure relational DBMS, however, everything has to be in 1FN, therefore eventual accessed virtual attributes has to be un-virtualized by made them explicit.
As final step nested lists has to be handled by creating surrogate values when the nested data is not accessed or to made the inner lists explicit when their data is required.

In chapter 4 the compilation rules from the normalized tree into the algebraic plan are shown. They follow the *loop lifting* approach in order to avoid the generation of avalanches of SQL queries due to query comprehensions.

In chapter 5 the resulting algebraic plan is given to Pathfinder to be optimized and to generate the SQL code out of it. Via ODBC it's then executed on the database and the result is stored in the heap.

Chapter 6 shows how to create the new LINQ-to-SQL provider, which will be used when a query against a database is performed.

Chapter 2

Software overview

LINQ is a general-purpose query facilities that allows a software developer to use a set of query operators to interact with any structured data source. LINQ has 2 approaches to query data: let the .NET Framework do the job or handle it to a data-source specific provider.In this chapter a brief description of LINQ is presented with a focus on the provider approach and the iQueryable interface.

The Pathfinder engine is used to optimize the generated algebraic plan representing the LINQ query and to generate the SQL code.

2.1 LINQ

LINQ is the acronym of *Language-INtegrated Query*. It is a set of extensions to the .NET Framework (released with the 3.5 version) to integrate a query language to the primary programming language (based on the .NET Framework), making much more easier for the developers to interact with structured data from many different in-memory sources, using a set of general-purpose standard query operators in a declarative way exposed to the user as the Standard Query Operator API. It works as a middle tier between data store and the language environment.

The only constrain to use LINQ to query a collection is that the data source needs to have its data encapsulated into objects, so for data source not in that form (like databases) the data must be mapped into the object domain before being queryable.

A LINQ query is a method-based query, so for many query operators the developer have to specify a function (which is called lambda expression) to perform projections, filtering and

other operations. Those functions can be written as named or anonymous methods, or as lambda expressions, the way C# 3.0 introduces to write delegates in a compact way. Those lambda expressions of course have to adhere to the method signature defined by a delegate type. The advantage of lambda expressions is that they can be compiled as either code or expression tree, which allows them to be processed at runtime by optimizers, translators, and evaluators.

The queries written using the LINQ query operators are always lazily evaluated and are executed either directly by the LINQ query processing engine inside the .NET Framework or via an extension mechanism. This second approach is based on handing the work to various data-source specific providers which either implement a separate query processing engine or translate the request to a Domain Specific Query Language to be executed on a separate data source (such as on a database server as SQL queries).

In both cases, the result is then returned as a collection of in-memory objects that can be enumerated as any other enumerable object, through functions as *foreach*. This is done to use less memory and allows the user to start processing the first values without waiting for the whole collection to be ready. For most of the operators, the enumeration will stop when a match is found, and then the delegates are evaluated. The subsequent object is enumerated only when it's retrieved. But for grouping operators, like GroupBy, OrderBy or the aggregate operators, an eager evaluation is required because those operators need all elements in the collection to work on.

The compilation approach is chosen depending on on which interface the source implements (either the *IEnumerable* for direct LINQ approach or the *IQueryable* for the provider approach). From a developer point of view it makes no difference because the queries are written in the same way, using the same operators, it just changes *how* they are handled at compile time.

New objects structures can be defined at the same moment of their initialization using anonymous types, defining a static shape of the object instead of defining it with both states and behaviors.

2.1.1 The iEnumerable interface

The set of query operators are defined as generic extension methods in the *IEnumerable* interface, which represent virtually any collection type in the .NET Framework. As a result, any

class that implements the *IEnumerable* interface has access to these methods and is therefore queryable. A concrete implementation of the query operators is provided in the *Sequence* class, but every developer can implement it's own implementation, or create and add his own set of domain-specific query operators, by simply adding extensions method to the *IEnumerable* interface. To ensure query composability each query operator returns an object implementing the same *IEnumerable* interface, so the output of one can be used as the input to another method.

Since a query on a collection implementing the *IEnumerable* interface is compiled directly into code and not in an expression tree, it doesn't allow any work on the query. Therefore no query optimization can be performed on it, so the query operators are executed in the original given order.

2.1.2 The iQueryable interface

As said above, in LINQ another interface, deriving from the *IEnumerable* interface, is defined: the *IQueryable* interface.
It has, as extensions method, the definition of almost the same set of query operators of the *IEnumerable* interface but they are implemented to build Expression object instead of giving the result. At compile time, by combining those objects together, an abstract syntax tree (AST) called *expression tree* is built. It represent the whole query, preserving its high level structure, as a tree of objects that can be parsed at run time by other code as any other data. The result of the parsing is then evaluated when the data is enumerated.
That means that instead of evaluating directly the query in the .NET Framework by translating it into code as it happens with collection implementing the *IEnumerable* interface, the query is at first converted into data, allowing other code to work with it.

In this approach a new tier is introduced between LINQ and the data source and it's a data-source specific implementation of the *IQueryable* interface called provider, which at run time consumes the generated expression tree, processes it and when the data is accesses, executes the generated code, returning the results in the same collection of in-memory objects as with the *IEnumerable* approach.

This approach allows the provider behind the scene to actually *work* with the query, optimizing it and translating it into a Domain Specific Query Language, for example into a SQL query to be executed against a database. That opens the door to adapting LINQ to querying virtually any data source by writing a new provider specifically for it. The high level structure

schema is shown in figure 2.1.

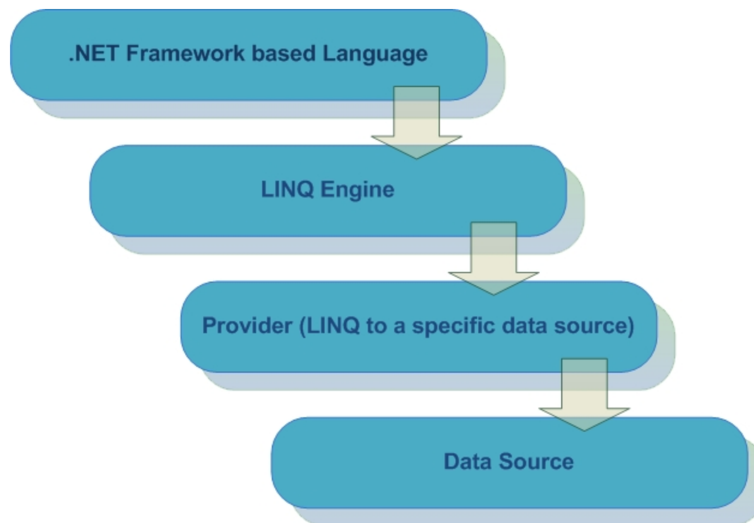


Figure 2.1: LINQ high level structure

The query written in the language environment is handed over to the LINQ engine that gives it to the right provider, basing the decision upon which is linked to the queried collection. The provider works on the given query and generates a *DynamicMethod* using the reflection APIs to produce *Common Intermediate Code* (CIL). This generated method is then executed on the data source when the results are accessed in the code.

For example, what happens behind the curtains using LINQ to SQL is that we write the query using the LINQ syntax and it's translated by the provider into SQL code that will be executed directly on the database, instead of moving all the data to the heap memory and then execute the required operation. LINQ to SQL is the only provider that use this approach. It is of course more logical, the DBMS is already optimized to do this type of operations and the data doesn't need to be copied into the heap with the risk of not having enough space there to evaluate query against a big database. In all the other cases the whole data is moved to the heap and then processed.

Inside the *IEnumerable* interface is defined the extension method *AsQueryable()* that allows an *IEnumerable* collection to be wrapped into an *IQueryable* collection, forcing the LINQ engine to use this provider based approach.

LINQ comes with already some basic providers like LINQ to Objects, which allows to query in-memory collections, LINQ to SQL, to query databases and LINQ to Xml, for xml

documents.

The focus of this thesis is to create a new LINQ-to-SQL provider which is *type-directed* instead of *data driven* as the Microsoft one.

2.2 The TPC-H Database

The example queries showed in this thesis are based on the database defined in the benchmark TPC-H.

2.3 Pathfinder

Pathfinder is a pure relational XQuery compiler. It allows a DBMS to process a XQuery query [1].

For the goal of this thesis some of its features are used. Our tree will be compiled into the Pathfinder dialect in order to use its optimizer [7], to obtain an heavily optimized plan, and its SQL Code generator [2] to translate the generated plan into SQL code that can be execute on any SQL:1999-capable DBMS.

Chapter 3

Normalization of the linq tree

The only thing our provider gets to work on is a LINQ expression tree, so we have of course to get to know it, how it looks like, how to traverse it (all of it, and as we will see it is not so immediate to achieve) and how to obtain all the information we need to generate the SQL code.

In the beginning of this chapter a more detailed introduction to the expression tree approach is given and a visitor to traverse a whole LINQ generated expression tree is described. This tree needs to be normalized in order to be compiled.

As first step, since an expression tree once is generated it cannot be modified, a new light weighted version has to be created using new implemented node type shaped to carry only the required informations in a much more readable way. To reduce the number of compilation rules our compiler has to handle, this new tree has to be stripped down of many redundant SQO (and their overloads) which will be substituted by equivalent combination of other SQO. Inside a LINQ query, due to the object mapping of the database, virtual attributes can be present because it's possible to navigate through associations using the dot notation. This makes the intermediate results and the query result not in first normal form, because a virtual attribute is not more atomic but is an entire object with its method and properties. Therefore virtual attributes needs to be *un-virtualized*, meaning a join is performed between the two linked table when a virtual attribute is accessed in order to have only one table which is in 1FN. The last step will make the complex LINQ structure, such as records and especially nested lists, map to the more simple structure that can be used in a pure relational database (namely tuples and tables) via the introduction of the built-in Box and UnBox operators.

At the end of this normalization phase the new AST will be ready to be compiled into SQL code.

3.1 Expression trees

An expression tree represents, through an in-memory data structure, a lambda expression or a query, preserving its high level structure. It's the way used by the .Net Framework to work with expressions: it doesn't generate directly executable code but represents their code as data that can be analyzed at run-time by other tools.

The compiler translates the expression in a series of method calls to the facilities offered by the name-space System.Linq.Expressions where the classes and interfaces to built language-level code expressions can be found. At run-time that code is executed and the expression tree generated. Each node in the expression tree represents an expression, for example a method call like a Select SQO or a binary operation such as x < y.

A developer can create it by hand, and so dynamic queries are possible or it's generated by the compiler from a lambda expression if its type is Expression<TDelegate> (meaning it has an expression body) or from a LINQ query on a collection that implements the IQueryable interface. In these cases the compiler doesn't translate the query into IL code that evaluates the query but into IL code that, at run-time, generates an AST out of it allowing other tools to work with it.

For example, this simple query extracts the name of each Customer, wrapping it inside a new user-defined anonymous object:

```
1  var query = db.CUSTOMER.Select(x => new { name = x.C_NAME } );
```

This query will be translated by the C# compiler into a set of nested calls to the static methods of the name-space System.Linq.Expressions to instantiate the various nodes of the expression tree.
In our example the first method call will be to create the method call Select node that will have 2 other method calls as parameters: to generate the node that represents the CUSTOMER table and the node representing the lambda expression for the right side. The body of the lambda expression node will contain more calls to build the new record structure and so on, until the whole query is represented in the expression tree. Using a visitor we can see how the expression tree is shaped in figure 3.1.

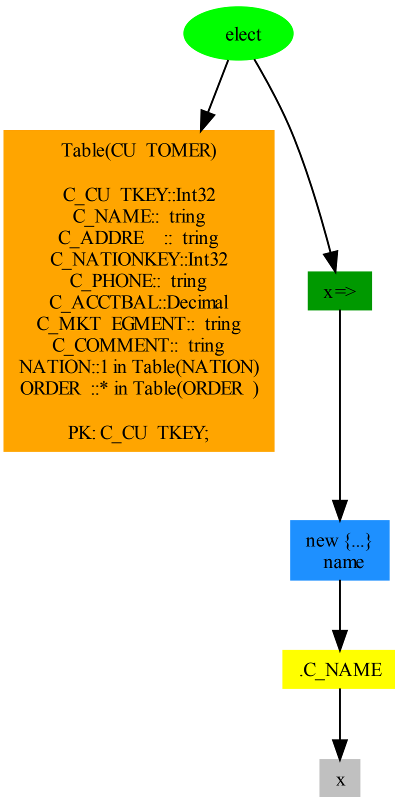


Figure 3.1: Structure of the expression tree for the query *varquery* = $db.CUSTOMER.Select(x => new\{name = x.C_NAME\});$

That's what made LINQ to SQL possible: the generated expression tree can be, at run time, analyzed by a tool that can work on it, in our case it can generate out of it SQL code. This SQL code can be then sent to the DBMS to be executed, and get the results from it.

In our case it will be the provider that consumes this generated expression tree, analyze it, modify it in order to be compiled and then translate it into SQL code that can be executed into any SQL:1999-capable RDBMS.

3.2 The new light-weighted AST

Once an expression tree is created it cannot be modified. That's not good for us because we have to tune the tree before compiling. As we will see in the first phase of this tuning we have to normalize some operators in order to have less compilation rules and that means substitute a node with one of another type or maybe with a new branch, formed maybe by 2 or 3 nodes. And also in the other phases that can easily happen as, for example, with the introduction of Box and UnBox operators.

The solution is the easier one can think about: traverse the tree and create a new instance of it. On the copy we can do of course whatever we want.

To take the most out of this step, we can think about something that will made our work in the next phases easier: create the new tree using nodes defined by us. First, getting the required information from the actual expressions node is not always easy, and most of this informations is useless to our goal. Adding to this the fact that add or modify expression nodes is not immediate because they are strongly typed since normally they are instantiated by the compiler at compile time so when no type-mismatch error can occur, the pros of having our own expression tree nodes are far more than the cons of writing a couple more classes to model our nodes. This will also produce a much more readable code and flexible tree, that, in future developments, can be easily adapted to new scenarios.

For the scope of this thesis we need to model only a few node types, in which all the informations we need about the operators handled until now can be stored. But it was noticed that for particular method calls (like Select) it's better to create their own class, instead of collapsing all of them into the Function Call class.

Each node type is modeled with its own class inheriting from a more general class as shown in the class diagram in figure 3.2. Those classes are presented here, with a brief description of their structure and scope.

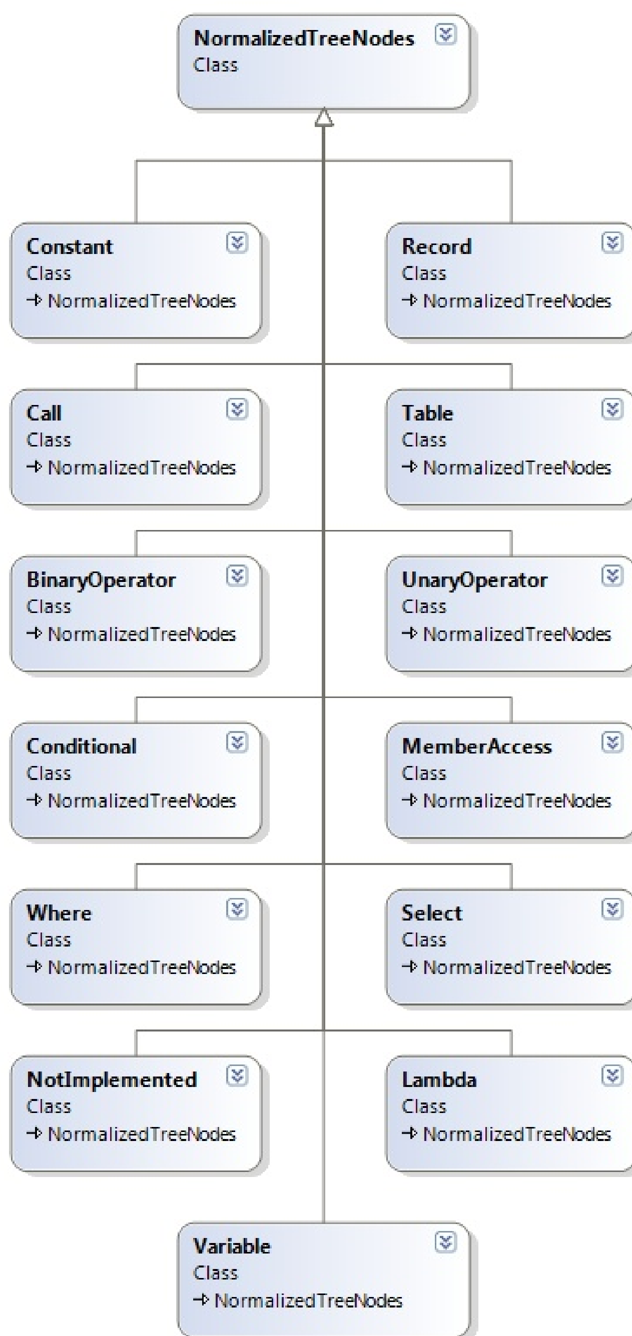


Figure 3.2: The new light-weighted AST class diagram

3.2.1 The AST super class

Description

All the subsequent classes inherit from a general super class called that contains general properties, helper and instantiation methods and enumerators specifying the node types.

Structure

There are enumerators to specifies some frequent used type names:

- nodeType: specifies the unique types of this new nodes; so far the types are:
  - BinaryOperator: it represents binary operators;
  - Function Call: it represents generic function calls;
  - Conditional: it represents conditional statements;
  - Constant: it represents constants;
  - Lambda: it represents lambda expressions;
  - MemberAccess: it represents member access;
  - NotImplemented: it is used for debug purpose, it represents a not yet modeled by this AST LINQ node;
  - Record: it represents New record structures;
  - Select: it represents the method calls Select;
  - Table: it represents database tables;
  - UnaryOperator: it represents unary operators;
  - Variable: it represents variables.

s properties we have:

- NodeType: it contains the type of the node;
- id: a unique numeric identifier used to maintain our tree a DAG.

Those properties are set when a specific node is created.

A static method it's defined for each node type, called to instantiate our new nodes.

3.2.2 Binary Operators

Description

The Binary Operator node represents binary expressions

Structure

The stored informations are:

- the name of the operator;
- the left and right side of the expression;
- the C# types of both sides.

3.2.3 Function Call

Description

The Function Call class represents a generic function call. It can have an arbitrary number of parameters. Some function calls, like Select, are modeled with their own class, because later they need to be threated differently.

Structure

The stored information are:

- the name of the function;
- the list of arguments;
- the prototype of the function.

3.2.4 Constant

Description

The Constant class represents a constant value. As seen in the original LINQ tree also the tables are represented through the same type of node. In our tree they are modeled as two different classes because they are two different concepts.

Structure

The stored informations are:

- the value;
- the type.

3.2.5 Lambda Expressions

Description

It represents a lambda expression. Due to our SQO normalization, we will have lambda expressions only on the Select operator.

Structure

The stored information are:

- the list of parameters it has;
- the body expression.

3.2.6 Member Access

Description

The Member Access node maps the member access we find in the original tree.

Structure

The stored information are:

- the name of the accessed member;
- the expression on which the member is accessed;

3.2.7 Not Implemented

Description

The Not Implemented node is used only for debug purpose. It's instantiated when a not yet handled LINQ node is found in the tree. It raises an exception in the compilation phase, warning the user with which type of LINQ node caused it.

Structure

It contains only the name of the LINQ that is not yet handled.

3.2.8 Record

Description

The Record node represents the declaration of a new anonymous type. In the original LINQ tree those information are to be found in the New nodes.

Structure

The stored information are:

- the list of parameters name;
- the list of the associated expressions.

3.2.9 Select

Description

The Select node represents the call to the Select function. Since later on will be compiled differently from the other method calls and will be frequently introduced during the normalization of the SQO, it's better to have it modeled by its own class.

Structure

The stored information are:

- the source expression, that must be of table type table
- the lambda expression, that must be of table type row.

3.2.10 Table

Description

The Table node represents a table of the database. In the original LINQ tree they are stored inside node of type Constant, but their concept is different, so they'll be shaped by their own class.

Structure

The stored information are:

- the table name
- the list of columns in a structure containing, for each column, its type and a boolean discriminating if it's part of the primary key or not;
- the list of associations in a structure containing, for each association, the name of the internal attribute, the name of the external table and relative referenced attribute and the cardinality

Also some helper methods to get various information are defined:

- to get the complete primary key;
- to get a list containing the column names;
- to get a list containing the column types;

3.2.11 Unary Operator

Description

The Unary Operator node represent unary operators.

Structure

The stored information are:

- the name of the operator;
- the expression on which is applied.

3.2.12 Variable

Description

The Variable node represents a variable.

Structure

The stored information are:

- the name of the variable

3.2.13 Walkabout

The whole LINQ expression tree must be traversed at run-time in order to create out of it a the new light-weighted version using the nodes defined in section 3.2. This new tree will contain all the information needed to generate the SQL code for the LINQ query as described in the structure of each node.

The traversing order must be postorder, meaning first all the children, starting from the left side, and then the root. When the root is visited, a corresponding node of our type must be instantiated, using the informations gathered in the LINQ node.

A case that needed to be taken into account is when an external variable is referenced inside the query. That will lead to a late binding due to the closure approach used by C# to handle outer reference inside expressions. In this case reflection should be use to obtain the value of the external value. In the case that the variable represents a query, its tree must be visited and merged with the one we are visiting.

At the end, so when the root of the LINQ tree is mapped to our nodes, we should have an exact copy of the entire (including also the eventual referenced outer queries) LINQ tree that we are able to modify as we want.

3.3 Too many SQO's overloads

In the *IQueryable* interface we can see that almost every SQO has many different overloads to handle different cases. Most of the times they differ only on the form of the output the user can specify through delegates (as for example the GroupBy operator) but others absolve different function depending on which overload is chosen.

If we give to our compiler the tree like it is now, we would have to write a compilation rule for each overload of every SQO, so a really really big set of rules. For example for the GroupBy operator we would have at least 4 different rules (8 if we support also a function for the equality comparer), for the Where other 2 and so on. Too many rules to write and, in future developments, to maintain.

However, there is a way to make our future work easier, much more elegant and flexible: adopting a compositional approach.

The idea is to have compilation rules only for a super-set of SQOs that can't be re-written as an equivalent combination of other SQOs, and that can be used as elements to re-write the other SQOs (and their overloads) in an equivalent sequence of those "base" SQOs.

Of course the tree will get a little bit bigger in that phase, but we will appreciate the advantages of that approach in the compilation phase, and also more when we want to optimize or rewrite the compilation rules in the future.

In this section we show all the handled SQO so far and, where needed, the rules that are used to normalize them. In this first version of the provider the overloads where the equality comparer method can be specified won't be taken into account. Those are not interesting for the scope of this thesis but can be easily handled by adding some normalization rules in this phase and some compilation rules later.

In the rules the operator $\Mapsto$ means *compiled into*. The result of the compilation is a subtree. For example:

$$e_1 \Mapsto e_1'$$

means that the subtree e_1 is compiled into a new subtree e_1' on which the normalization process is already applied.

3.3.1 Newly introduced operators

Some new operators are used in this normalization phase, because they are more efficient and reusable than the original LINQ ones. Here a brief description is given in order to help the reader understand them.

FILTER

The FILTER operator consumes two positional aligned lists: a list contains the elements of the sequence, the other contains a list of corresponding boolean values. Using the boolean list, the FILTER operator infers the resulting sequence, containing only the elements which the corresponding boolean value is *true*.

$$[true, false, true].FILTER([4, 8, 15]) = [4, 15]$$

GROUP

The GROUP operator performs a grouping based on two positional aligned lists: a list contains the key values and the other the elements. The elements of the source list are grouped based on the corresponding key in the key list.

$$[k1, k2, k1].GROUP([4, 8, 15]) = [(k1, [4, 15]), (k2, [8])]$$

FLATTEN

The FLATTEN operator consumes a list of lists and returns the flatten version of it.

$$[[4, 8], [15, 16]].FLATTEN() = [4, 8, 15, 16]$$

3.3.2 Aggregates

The aggregates operators apply different functions on a numeric column and returns a single numeric value.

We consider the following aggregate operators:

- Average: the Average operator returns the average of the elements of the column;
- Max: the Max operator returns the maximum value of the elements of the column;
- Min: the Min operator returns the minimum value of the elements of the column;

- Sum: the Sum operator returns the sum of the elements of the column.

They have many overloads which are based on the type of the source column but those doesn't need to be handled because for the compiler it makes no difference, it handles them in the same way as long as the type is specified.
What we have to normalize in this phase are the overloads where we also have a delegate to project out the column on which the aggregate operator will be applied.

1. $e_1.Agg()$: in that case the source must be a single numeric column on which the aggregate operator will be applied.

   It can't be rewritten as a combination of other SQOs, so the rule will be just a dispatch rule:

$$\frac{e_1 \mapsto e_1'}{e_1.Agg() \mapsto e_1'.Agg()}$$

2. $e_1.Agg(v \Rightarrow e_2)$: in this other case we have a delegate that filters the source in order to project out the column on which the aggregate operator will be applied.

   The normalized form reuses the previous overload but applied on the original source filtered by the specified delegate.

$$\frac{e_1 \mapsto e_1' \qquad e_2 \mapsto e_2'}{e_1.Agg(v \Rightarrow e_2) \mapsto e_1'.Select(v \Rightarrow e_2').Agg()}$$

3.3.3 All

The All operator determinate if all the elements in the source satisfy a given condition.

It can be normalized as the count of the original source filtered by the negation of the given condition, in order to have the number of how many elements don't satisfy the initial condition. If the number of these elements is 0, then no element satisfy the negate form of the condition so it means that they all satisfies the original one and true is returned. If the count is greater than 0, then false is returned.

$$\frac{e_1 \mapsto e_1' \qquad e_2 \mapsto e_2'}{e_1.All(v \Rightarrow e_2) \mapsto e_1'.FILTER(e_1'.Select(not(v \Rightarrow e_2'))).Count() == 0}$$

3.3.4 Any

The Any operator checks if the source contains any elements.

It has one overload where a filtering delegate can be specified.

1. $e_1.Any()$: returns false if the source is an empty list, true otherwise.

   It's equivalent to count the elements of the source and check if it's greater than 0. The result of the condition matches the result of the Any operator.

$$\frac{e_1 \;\mapsto\; e_1'}{e_1.Any() \;\mapsto\; e_1'.Count() > 0}$$

2. $e_1.Any(v \;\Rightarrow\; e_2)$: returns false if the source, filtered by the given condition, is an empty list, true otherwise.

   In this case the normalized form is almost the same as above but first the source is filtered through the given predicate. It's important to notice that the source is filtered using the normalized version of the Where operator according to the rule 3.3.16.

   It can be normalized as:

$$\frac{e_1 \;\mapsto\; e_1' \qquad e_2 \;\mapsto\; e_2'}{e_1.All(v \;\Rightarrow\; e_2) \;\mapsto\; e_1'.FILTER(e_1'.Select(v \;\Rightarrow\; e_2')).Count() > 0}$$

3.3.5 Concat

The Concat operator concatenates two sequences that have the same structure into one sequence. Contrary to the original LINQ-to-SQL provider, ours can handle every type of structure, including the hierarchical ones.

It doesn't have any overloads and it can't be rewritten in a simpler way, so we just have a dispatch rule:

$$\frac{e_1 \;\mapsto\; e_1' \qquad e_2 \;\mapsto\; e_2'}{e_1.Concat(e_2) \;\mapsto\; e_1'.Concat(e_2')}$$

3.3.6 Count

The Count operator returns the number of elements in a given sequence.

It has one overload where a delegate to filter the source can be expressed.

1. $e_1.Count()$: returns the count of the original sequence.

   It can't be rewritten as a combination of other SQOs, so the rule will be just a dispatch rule:

$$\frac{e_1 \mapsto e_1'}{e_1.Count() \mapsto e_1'.Count()}$$

2. $e_1.Count(v \Rightarrow e_2)$: returns the count of the original sequence filtered by the given function.

   It can be normalize by applying the filtering delegate to the original source before counting:

$$\frac{e1 \mapsto e1' \qquad e2 \mapsto e2'}{e1.Count(v \Rightarrow e2) \mapsto e1'.FILTER(e_1'.Select(v \Rightarrow e2')).Count()}$$

3.3.7 Distinct

The Distinct operator returns the source sequence where duplicated tuples are eliminated.

It has no overloads and can't be rewritten in a simpler way, so we'll have again only a dispatch rule:

$$\frac{e_1 \mapsto e_1'}{e_1.Distinct() \mapsto e_1'.Distinct()}$$

3.3.8 ElementAt

The ElementAt operator returns the element of the source at a given position. The actual LINQ-to-SQL provider doesn't support that operator, but the provider we are developing can, because it has a strict order among the results, due to the *looplifting* approach.

It has no overloads and can't be rewritten in a simpler way, so we'll have again only a dispatch rule:

$$\frac{e_1 \ \mapsto\ e_1' \qquad e_2 \ \mapsto\ e_2'}{e_1.ElementAt(e_2) \ \mapsto\ e_1'.ElementAt(e_2')}$$

3.3.9 First

The First operator returns the first element of a given sequence.

It has one overload where a filtering delegate can be specified.

1. $e_1.First()$: in this case it returns the first element of the original sequence.

   It can be normalized using the ElementAt operator to get the element in the first position:

$$\frac{e_1 \ \mapsto\ e_1'}{e_1.First() \ \mapsto\ e_1'.ElementAt(1)}$$

2. $e_1.First(v \ \Rightarrow\ e_2)$: in this case it returns the count of the original sequence filtered by the given function.

   It can be normalize as above with just the application of the filtering delegate on the original sequence:

$$\frac{e1 \ \mapsto\ e_1' \qquad e_2 \ \mapsto\ e_2'}{e_1.First(v \ \Rightarrow\ e_2) \ \mapsto\ e_1'.FILTER(e_1'.Select(v \ \Rightarrow\ e_2')).ElementAt(1)}$$

3.3.10 GroupBy

The GroupBy operator groups the element of a sequence based on a key selector delegate. To normalize it, the new GROUP operator is used.

It has 4 different overloads, on which delegates to shape the form of the type of the elements in each group can be defined. In all overloads a key selector delegate representing the group condition must be specified.

- $e.GroupBy(x \Rightarrow e_k)$: in this case we only have the key selector delegate. It returns only the value of the keys.

  It can be normalized using the GROUP operator. The element list is the original source, while a Select, based on the key selector function, projects out the keys to form the key list.

$$\frac{e \mapsto e' \qquad e_k \mapsto e'_k}{e.GroupBy(x \Rightarrow e_k) \mapsto e'.GROUP(e'.Select(x \Rightarrow e'_k))}$$

- $e.GroupBy(x \Rightarrow e_k,\ x \Rightarrow e_v)$: in this case a element selector is invoked to obtain a result element.

  The GROUP works on the same key list as above, while the value list is obtained by filtering the original source by the specified element selector delegate.

$$\frac{e \mapsto e' \qquad e_k \mapsto e'_k \qquad e_v \mapsto e'_v}{e.GroupBy(x \Rightarrow e_k,\ x \Rightarrow e_v) \mapsto e'.Select(x \Rightarrow e'_v).GROUP(e'.Select(x \Rightarrow e'_k))}$$

- $e.GroupBy(x \Rightarrow e_k,\ (x_k,\ x_v) \Rightarrow e_r)$: in this case the result selector parameter is used to obtain a result value from each group and its key.

  The GROUP operates on the key list as before, but this time the value list is not filtered so the entire tuple is bounded to its key.

  To obtain the type specified by the result selector, this intermediate result needs to be massaged in order to have the variable x_k mapped to the key values and the variable x_v mapped to the results. Using a Select operator where a new generated q variable binds to

an entire tuple of our result, those mapping are done. Inside the result selector delegate, the variable x_k will be substituted by $q.Key$, which represents the key values, and the variable x_v is substituted by the whole q, allowing the user to access the whole tuple represented by a key.

$$\frac{e \mapsto e' \qquad e_k \mapsto e'_k \qquad e_r \mapsto e'_r}{\begin{array}{c} e.GroupBy(x \Rightarrow e_k,\ (x_k,\ x_v) \Rightarrow e_r) \mapsto \\ e'.Select(x \Rightarrow e'_k).GROUP(e').Select(q \Rightarrow e_r[x_k/q.Key][x_v/q]) \end{array}}$$

- $e.GroupBy(x \Rightarrow e_k,\ x \Rightarrow e_v,\ (x_k,\ x_v) \Rightarrow e_r)$: in this overload both the result selector and the element selector are specified. The element selector parameter is used to project the elements of each group, and the result selector parameter is used to obtain a result value from each group and its key.

  The normalization is almost the same as the overload with only the result selector, with the only difference that the elements list of the GROUP operator is the source filtered by the element selector instead of the original source.

$$\frac{e \mapsto e' \qquad e_k \mapsto e'_k \qquad e_r \mapsto e'_r}{\begin{array}{c} e.GroupBy(x \Rightarrow e_k,\ (x_k,\ x_v) \Rightarrow e_r) \mapsto \\ e'.Select(x \Rightarrow e'_k).GROUP(e'.Select(x \Rightarrow e'_v)).Select(q \Rightarrow e_r[x_k/q.Key][x_v/q]) \end{array}}$$

3.3.11 Select

The Select operator projects all elements of a sequence into a desired form.

It has two overloads, but for now we don't handle the positional Select. It can't be rewritten in a simpler form so we have again just a dispatch rule:

$$\frac{e_1 \mapsto e'_1 \qquad e_2 \mapsto e'_2}{e_1.Select(v \Rightarrow e_2) \mapsto e'_1.Select(v \Rightarrow e'_2)}$$

3.3.12 SelectMany

The SelectMany operator projects all element of a sequence into a desired type and combines the resulting sequences into one sequence with elements of the same type.

It has 4 overloads, but two of them are based a positional Select that it's not yet handled.

1. $e_1.SelectMany(x \Rightarrow e_2)$: the basic version, described above.

   It can be normalized as a Select on which result the new FLATTEN operator is applied.

$$\frac{e_1 \Mapsto e_1' \qquad e_2 \Mapsto e_2'}{e_1.SelectMany(x \Rightarrow e_2) \Mapsto e_1'.Select(x \Rightarrow e_2').FLATTEN()}$$

2. $e_1.SelectMany(x \Rightarrow e_2, (x, y) \Rightarrow e_3)$: after projecting, a result selector function is applied to every element. The intermediate sequences are then combined into a single, one dimensional sequence and returned.

   It can be normalized as above with the difference that the result selector is applied to the projection delegate.

$$\frac{e_1 \Mapsto e_1' \qquad e_2 \Mapsto e_2' \qquad e_3 \Mapsto e_3'}{\begin{array}{c} e_1.SelectMany(x \Rightarrow e_2, (x, y) \Rightarrow e_3) \Mapsto \\ e_1'.Select(x \Rightarrow e_2'.Select(y \Rightarrow e_3')).FLATTEN() \end{array}}$$

3.3.13 Single

The Single operator returns the only element of a sequence as a value and not as an array with only one element.

It has one overload where a filtering function can be specified.

1. $e_1.Single()$: returns the only element of the original sequence.

   It can't be normalized and so we have a dispatch rule:

$$\frac{e_1 \Mapsto e_1'}{e_1.Single() \Mapsto e_1'.Single()}$$

2. $e_1.Single(v \Rightarrow e_2)$: returns the only element of the original sequence, on which the filter is applied.

   It can be normalized with the introduction of a filtering operator on the original source and then using the above overload.

$$\frac{e_1 \Mapsto e_1' \qquad e_2 \Mapsto e_2'}{e_1.Single(v \Rightarrow e_2) \Mapsto e_1'.FILTER(e_1'.Select(v \Rightarrow e_2')).Single()}$$

3.3.14 Skip

The Skip operator skips the first *n* elements and returns the remaining elements of a sequence.

It doesn't have any overloads and we can't rewrite it, so we have again only a dispatch rule:

$$\frac{e_1 \Mapsto e_1' \qquad e_2 \Mapsto e_2'}{e_1.Skip(e_2) \Mapsto e_1'.Skip(e_2')}$$

3.3.15 Take

The Take operator returns the first *n* elements of a sequence.

It doesn't have any overloads and we can't rewrite it, so we have again only a dispatch rule:

$$\frac{e_1 \Mapsto e_1' \qquad e_2 \Mapsto e_2'}{e_1.Take(e_2) \Mapsto e_1'.Take(e_2')}$$

3.3.16 Where

The Where operator filters the values of a sequence based on a predicate.

It can be normalized using the FILTER operator applied on the original list. The boolean list is calculated by applying the condition delegate to the original sequence:

$$\frac{e_1 \Mapsto e_1' \qquad e_2 \Mapsto e_2'}{e_1.Where(v \Rightarrow e_2) \Mapsto e_1'.FILTER(e_1'.Select(x \Rightarrow e_2'))}$$

3.4 Go back to 1 Normal Form

As said before, in LINQ the tables are encapsulated and threated as objects. This allows the developer to use the facilities of the object world to interact with them, delegating the back end to translate these high level facilities back to a lower level that can be compiled.

In our case, to be compiled, the tables and the intermediate results inside the query must be in the first normal form (1FN), that it means that there can't be any non atomic values inside a column and each row must contain the same columns.

While the second part of the 1FN rule, that forces all the row to have the same columns, is ensured by the fact that the tables are represented as objects, so they have a static structure that must be the same for all the tuples and can't be changed, when there is a one-to-many relation or a one-to-one relation inside a table the first part can be broken.

This happens because a foreign key inside a table is represented in the object world as an instance or a collection (in the case of a one-to-many relation) of the object representing the referenced table. This breaks the rule because that attribute is not anymore an atomic value as requested in the rule but an object, that can have its own properties and methods. The developer can therefore access them using the dot notation as with any other object. So, in a sense, the attributes of the external table become virtual attributes of the table referencing it.
It's to be noted that this can be recursive: also a referenced table can have foreign keys that can be accessed remotely, allowing the developer to navigate through the tables.

For example in LINQ is possible to *navigate* through the tables using the associations:

```
1 var query = db.CUSTOMER.Select(v => new { name = v.C_NAME, region = v.
      NATION.REGION.R_NAME });
```

This query returns the name of each customer with its region name. This information about its region is accessed by navigating from the table CUSTOMER to the table REGION by using the external attributes NATION and REGION.

What needs to be done when a virtual attribute is accessed it needs to be *un-virtualized*. The two linked tables must be merged together, to obtain a single table where all the attributes are explicit and there are no more virtual ones, going back to be in first normal form.

3.4.1 Un-virtualizing

To merge the two linked table a join operation must be performed on the foreign keys. In a practical way, what needs to be done is to visit again the tree and, when a virtual attribute is accessed, to insert this new join operation in the expression tree right before the attribute is accessed, making the query accessing not anymore the virtual attribute but the un-virtualized one. This operation must be inserted only when a virtual attribute is accessed because it's a costly operation, and it's useless to perform it every time a table has an external reference. If this control is not done, this will lead, due to the possible aforementioned recursion of foreign key, to actually perform a join between all the tables of the database.

To execute the join some information are needed: the name of the tables involved and the keys on which the join is performed. Those information can easily be found when the table node in the tree is visited. The problem is that we will need those information only when the virtual attribute is accessed, so we need an environment where those information can be stored and retrieved when needed.

3.4.2 The TableMap environment

The TableMap environment contains the table information needed to perform a join between teo tables when a virtual attribute is accessed. This environment is needed because those information are available only on the node representing the table and not when a member access is performed. In order to have this information available when needed, they must be saved in an environment that will be created and modified when the tree is visited to transform the tables to the first normal form.

This environment is not so easy to model because it has to overcome various obstacles concerning again the object world, and specifically variable mappings and their scope.

In LINQ to work on a set of tuples (that can be either a table or an intermediate result), the operator bounds, at each iteration, a tuple to a variable defined as a parameter of its lambda expression and then processes it. Inside the lambda expression this variable represent the source as an instance of the source's type, so the dot notation is used to access the attribute inside it. This lead to have our source mapped to this variable, so if our source it's a table it will be referenced using the variable name and not through its own name.

In the example, we want to project out the name of the nation of each customer inside the CUSTOMER table. This information is stored in the attribute N_NAME of the table NATION, which is linked to the table CUSTOMER via its attribute NATION:

```
1  var query = db.CUSTOMER.Select(v => v.NATION.N_NAME);
```

At each iteration, the variable *v* is bounded to a tuple of the table CUSTOMER, and then, using the dot notation, the wished attribute is accessed. The variable *v* is an object instance of the type representing the table CUSTOMER of our database.
The environment must therefore keep trace of these mappings between tables and variables because inside the query all the tables are mapped to variables, and so they are the only names we can use to retrieve the information of a table.

This will bring another problem to the table: in every programming language a variable has its own scope inside which can be accessed and that allows other variables outside its scope to have the same name.

For example that happens in this query, where we filter the customers based on their nation:

```
1  var query = db.CUSTOMER.Select(v => v.NATION).Where(v => v.N_NAME.Equals("
       Germany"));
```

The variable *v* inside the Select operator refers to the table CUSTOMER, but inside the Where operator another variable with the same name refers to the result of the Select operator, so it represents the projection of the virtual attribute NATION of the table CUSTOMER.
So the environment must take into account also that problem, and when a virtual attribute of a table is accessed and the information to perform the Join are needed, it must give back the right informations based on the actual scope of the variable.

It can also happen, as shown in the next example, that a variable that is bounded to a certain table will be bounded to another variable. This query returns the same result as the above example, only with a different form of the resulting objects:

```
1  var query = db.CUSTOMER.Select(v => new { table = v }).Where(x => x.table
       .NATION.N_NAME.Equals("Germany"));
```

In this case the variable v refers to the table CUSTOMER. Inside the lambda expression of this operator a new anonymous type is created where a variable named *table* refers to the variable v, so to the whole CUSTOMER table. In the Where operator, the variable x is of the new defined anonymous type. Inside it, its member *table* can be accessed and can be used to reach the virtual attribute NATION of the table CUSTOMER and apply the filter based on the nation name.
Also that has to be handled by permitting a variable to map either to table infos (when it reference a table directly) or to map to another variable. So nesting mappings have to be allowed.

At the end our environment must allow a variable to map either table information or another variable, leading it to have nested reference, and must respect the scope of the variables declared in the query.

In the environment, called Γ, the mapping between variables and table infos are to be saved.

The informations of a table that need to be saved to execute a join based on the foreign keys are:

- the name of the tables involved;
- the name of the attributes involved;
- the cardinality of the relation: one-to-many or one-to-one

3.4.3 The Mapping Rules

A set of rules is needed to correctly feed, maintain and use the information inside the environment. Those rules are compositional, so can be applied in every order leading always to the correct result, and will be applied when the tree is visited. A rule for each type of node of the newly created AST must be written. Some of those are however just dispatch rules because they don't bring any useful information and they don't need any information contained into the environment.

The Table rule is the base for feeding the environment with table information. The Select and Record rules modify the environment in order to maintain correct mappings between variables and information, because inside them new variable mappings can be defined. The Variable rule just consume informations inside it, in order to get information about what the

variable refers to. The Member Access rule use the environment to see if the Join operation must be inserted when a member is accessed.

In the rules the operator $\Mapsto$ means *compiled into* in the context of the given Γ environment. The result of the compilation is a structure containing an un-virtualized subtree and a new Γ environment. At the beginning the Γ environment is empty.

For example:

$$\Gamma \vdash e_1 \Mapsto (e_1', \Gamma')$$

means that the subtree e_1 is compiled in the context of Γ into a new subtree e_1', on which the normalization process is applied, and a new Γ' environment containing the information gather during the visit of the e_1 subtree.

Table

This rule saves the table informations contained in the node into the environment. Since the node directly represents the table no variable mapping has to be specified.
A table can have an arbitrary number of virtual attributes whose information need to be saved into the environment. The information are the name of the table and its eventual associations with all the parameters need to join the two tables, such as the name of the external table, the foreign keys and the cardinality of the relation.
If the table doesn't contain any virtual attribute, only its name it's inserted in the environment.

This rule returns a new environment containing the information of the table.

$$\frac{}{\Gamma \vdash Table \Mapsto (Table, TableInfo)}$$

Select

When a Select node is accessed, the source of the operator is evaluated first because those information define what the Select operator can access. The returned information of the source are then nested in a new environment where the parameter name of the Select's lambda expression

maps to the environment gave back from the source. This is done because inside the lambda expression this parameter name is used to access the source structure.
This new environment is given to evaluate the lambda expression.

This rule returns the environment generated from the visit of the lambda expression, which represents what can be accessed after the Select operator is applied.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \mapsto\!\!\Rightarrow (e_1', \Gamma') \\ \{v \mapsto \Gamma'\} \vdash e_2 \mapsto\!\!\Rightarrow (e_2', \Gamma'')\end{array}}{\Gamma \vdash e_1.Select(v \Rightarrow e_2) \mapsto\!\!\Rightarrow (e_1'.Select(v \Rightarrow e_2'), \Gamma'')}$$

Record

The Record node defines the structure of a new type. Inside it new members can be defined. Each of them has an expression that defines its content. Therefore every expression must be visited and the returned environment will be inserted in the global environment mapped to the referred member name. This is done to maintain the scope of each name.

$$\frac{\Gamma \vdash e_i \mapsto\!\!\Rightarrow (e_i', \Gamma_i')|_{i=1,\ ...,\ n}}{\Gamma \vdash new\begin{Bmatrix} f_1 & = & e_1 \\ & \vdots & \\ f_n & = & e_n \end{Bmatrix} \mapsto\!\!\Rightarrow \left(new\begin{Bmatrix} f_1 & = & e_1' \\ & \vdots & \\ f_n & = & e_n' \end{Bmatrix}, \begin{Bmatrix} f_1 & \mapsto\!\!\Rightarrow & \Gamma_1' \\ & \vdots & \\ f_n & \mapsto\!\!\Rightarrow & \Gamma_n' \end{Bmatrix}\right)}$$

Variable

The Variable rule consumes the information inside the environment. It un-nest eventual mapping information inside the environment to create the actual environment.

There can be 2 possibilities:

- The environment contains an entry for the variable v. This mapping can be either a nesting value or directly table information.

  Those information are then given back as the new environment.

  - The environment contains a mapping directly to table information. This happens when the variable maps directly a table:

$$\frac{\Gamma \Rrightarrow \{\ldots,\ v\ \mapsto\ Table\ Information,\ \ldots\}}{\Gamma\ \vdash\ v\ \Rrightarrow\ (v,\ Table\ Information)}$$

– The environment contains a mapping to an another environment. This happens when the variable maps to another variable:

$$\frac{\Gamma \Rrightarrow \{\ldots,\ v\ \mapsto\ \Gamma',\ \ldots\}}{\Gamma\ \vdash\ v\ \Rrightarrow\ (v,\ \Gamma')}$$

- The environment contains no entry for the variable v. This happens when the variable maps, for example, to a constant column.

  As new environment an empty set is therefore returned.

$$\frac{\Gamma \Rrightarrow \{no\ map\ for\ v\ in\ \Gamma\}}{\Gamma\ \vdash\ v\ \Rrightarrow\ (v,\ \emptyset)}$$

Member Access

The Member Access rule uses the information contained in the environment in order to decide whether a virtual attribute of a variable is accessed or not. In the first case the Join must be inserted to transform the table in 1FN, while in the second case no operation needs to be done.

There can be 3 possibilities that lead to different rules:

- The environment contains an entry for the member m that maps the name directly to table information. That means that a virtual attribute is accessed and a Join must be inserted.

  This rules returns a new environment containing the table informations of the external table because it's what can be accessed after the Join.

$$\frac{\begin{array}{c}\Gamma\ \vdash\ e\ \Rrightarrow\ (e',\ \Gamma')\\ \Gamma' \Rrightarrow \{\ldots,\ m\ \mapsto\ O :: TableInfo,\ \ldots\}\end{array}}{\Gamma\ \vdash\ e.m\ \Rrightarrow\ (Normalized\ Join\ based\ on\ e',\ TableInfo\ of\ the\ other\ relation)}$$

For ease of readability, the description of the Join branch inserted by this rule is presented in details in section 3.4.4.

- The environment contains an entry for the member m that maps to another environment, meaning that it's a nested mapping.

  In this case the nested information are given back as the new environment.

$$\frac{\Gamma \vdash e \Rrightarrow (e', \Gamma') \quad \Gamma' \Rrightarrow \{\ldots, m \mapsto \Gamma'', \ldots\}}{\Gamma \vdash e.m \Rrightarrow (e'.m, \Gamma'')}$$

- The environment contains no entry for the member m. That means that it's already an explicit attribute and doesn't need to be un-virtualized:

$$\frac{\Gamma \vdash e \Rrightarrow (e', \Gamma') \quad \Gamma' \Rrightarrow \{no\ map\ for\ m\ in\ \Gamma'\}}{\Gamma \vdash e.m \Rrightarrow (e'.m, \emptyset)}$$

Constant

The Constant is just a dispatch rule. Since a constant is a static explicit value no information contained in the given environment is needed.

Inside it no new variable mapping is introduced and therefore the returned environment is empty.

$$\frac{}{\Gamma \vdash c \Rrightarrow (c, \emptyset)}$$

Binary

In the Binary rule both operand must have empty environment because they are both explicit values and not variables.

The rule returns an empty set because the result is a single column with the result of the operation.

$$\frac{\Gamma \vdash e_i \mapsto (e_i', \emptyset)|_{i=1,\,2}}{\Gamma \vdash e_1 \circledast e_2 \mapsto (e_1' \circledast e_2', \emptyset)}$$

Unary

Similar to the Binary rule, in the Unary rule the body expression has an empty environment and an empty environment is returned.

$$\frac{\Gamma \vdash e_1 \mapsto (e_1', \emptyset)}{\Gamma \vdash \circledast(e_1) \mapsto (\circledast(e_1'), \emptyset)}$$

Conditional

The Conditional rule is a dispatch rule. The condition and the choices expressions are evaluated independently using the given environment.

The rule returns the environment gave back by either the true or the false branch. They must be of the same type, therefore they will have the same environment so it makes no difference which environment is returned.

$$\frac{\begin{array}{c}\Gamma \vdash e_i \mapsto (e_i', \Gamma_i)|_{i=1,\,2,\,3} \\ \Gamma_2 \equiv \Gamma_3\end{array}}{\Gamma \vdash e_1\,?\,e_2 : e_3 \mapsto (e_1'\,?\,e_2' : e_3', \Gamma_2)}$$

Lambda

The Lambda rule performs just a dispatch of its body expression because after the SQOs normalization the only operator that has a Lambda expression is the Select rule. Therefore the introduction of the new variable is handled inside the Select rule.

The rule returns the environment obtained from its body expression, which shapes what can be accessed after it.

$$\frac{\Gamma \vdash e_1 \mapsto (e_1', \Gamma')}{\Gamma \vdash (v, \ldots, v_n) \Rightarrow e_1 \mapsto ((v_1, \ldots, v_n) \Rightarrow e_1', \Gamma')}$$

Function Call

The function calls, such as GROUP, FILTER, Take and so on, are gathered in just one rule. It's again a dispatch rule because no information from the environment are needed or modified.

However for some functions the environment returned is different: some of them modifies the scope of what can be accessed after their application, others not. Therefore for the functions that don't modify it, the environment generated by the evaluation of the source must be returned, for the others an empty environment is given back.

The environment is modified by functions that modify the structure of the source, such as the aggregate operators, because they create out of the source a new column and return it. The other function calls that just modify the content of the source and not its structure don't affect the environment structure.

$$\frac{\Gamma \vdash e_i \Mapsto (e_i',\ \Gamma_i')|_{i=1,\ \ldots,\ n}}{\Gamma \vdash f(e_i,\ \ldots,\ e_n) \Mapsto \left(f(e_i,\ \ldots,\ e_n), \begin{cases} \emptyset & \text{if } f \subset \Phi \\ \Gamma_1' & \text{if } f \subset \Lambda \end{cases}\right)}$$

$\Phi \in \{Avg, Sum, Max, Min, Count\}$

$\Lambda \in \{All, Any, Concat, Distinct, Drop, ElementAt, FILTER, First, GROUP, Select, Single, Take\}$

3.4.4 The Join branch

To perform the Join operation a new branch must be inserted in the tree. This branch represents the set of operations needed to merge the two tables. It will replace the member access when it accesses a virtual attribute as shown in the rule for Member Access in section 3.4.3.

As said above, the information needed to perform a Join between two referenced table are the name of the tables involved, the attributes forming the foreign key and the cardinality of the relation: all information that are saved in the environment.

The Join is composed by a Where operator applied on the external table with the lambda expression resembling the Join condition. This condition have to contain an equality comparison between each pair of keys in order to bind the right tuples.

At an high level the branch looks like:

$$e_1.Where(q \Mapsto e_2);$$

where e_1 represents the external table and e_2 the join condition.

As exposed in section 3.4, in LINQ each table must be mapped to a variable before its attribute can be accessed. Therefore a new variable representing the external table must be created and used to iterate through it. This new variable will be the parameter of the lambda expression of the Where operator and will be used to access the attributes of the external table. In the high level structure example it's called q.

The Join Condition

The Join condition is expressed as the lambda expression of the Where operator. Its parameter name is automatically generated and will map the external table, while in the expression the key comparisons will take place. The number of attributes that composes the key can lead to different scenarios.

The simplest case is when the key is composed by just one attribute pair: in that case inside the lambda expression there will be only the equality comparison between them.

For example, the foreign key between the tables Customer and Nation is composed only by one attribute, Nation.N_NationKey that refers to Customer.C_NationKey:

```
1  var query = db.CUSTOMER.Select(v => new { name = v.C_NAME, nation = v.
       NATION });
```

Therefore the Join condition to normalize the nation member access will be composed only by:

```
1  v.N_NationKey == q.C_NationKey
```

where the variable q is the newly generated parameter of the lambda expression which refers to the external table Nation. In that case the Join condition will be formed only by a binary operator that has as children the member access to the external keys.

We remark that, in order to handle the aforementioned possible nesting of variable mappings, the variable v, that in the example represents the internal table, it's the whole (already normalized by the environment mapping rules shown in 3.4.3) expression of the member access.

If we introduce a variable nesting in the above example, like this:

```
1  var query = db.CUSTOMER.Select(v => new { t = v}).Select(v => new { name =
       v.t.C_NAME, nation = v.t.NATION.N_NAME});
```

The Join Condition will be:

```
1  v.t.N_NationKey == q.C_NationKey
```

The variable nesting has to be maintained in order to access the right key.

A more complex scenario is where the key is composite, so formed by more than one attribute pair. This will lead to have an AND operator between each key comparison, because all the keys must match at the same time.

That happens, for example, in the reference between PartSupp and LineItem, where we have two key pairs: PartSupp.PS_SuppKey that refers to LineItem.L_SuppKey and PartSupp.PS_PartKey that refers to LineItem.L_PartKey.

```
1  var query = db.PARTSUPP.Select(v => new { key = v.PS_PARTKEY, lineitems =
       v.LINEITEM });
```

In this case the Join condition will contain also another binary operator to express the AND condition:

```
1  v.PS_SuppKey == q.L_SuppKey && v.PS_PartKey == q.L_PartKey
```

This will lead to create a more complex subtree for the Join condition due to the introduction of this Binary operator between the key matchings. The branch will have as root the binary AND operator whose kids are the comparison of the keys.

Formally, if we have n key pairs a subtree described by this rule must be inserted to represent the Join condition:

$$\Big\{\{\{[k_{1a} \ == \ k_{1b}] \ \&\& \ [k_{2a} \ == \ k_{2b}]\} \ \&\& \ [k_{3a} \ == \ k_{3b}]\} \ldots \ \&\& \ [k_{na} == k_{nb}]\Big\}$$

So each binary AND operator to add a new key comparison must be put on top of the existing comparison tree, meaning it has as left side the subtree involving all the previous key match operations and as right side the new key comparison.

The Cardinality

Also the cardinality of the relation has to be taken into account to shape the Join branch. This leads to have 2 possible options:

- One-to-Many: the branch that will perform the Join is:

$$e_1.Where(q \mapsto e_2);$$

where e_1 represents the external table and e_2 the join condition.

- One-to-One: the branch that will perform the Join is:

$$e_1.Where(q \mapsto e_2).Single();$$

where e_1 represents the external table and e_2 the join condition.

The Single operator is applied because it's a one-to-one relation so each tuple of the internal table maps to exactly one tuple of the external table. The Single operator will prevent the result to be a list with only one element.

The cardinality many-to-many is not possible because in a database it's normalized using an intermediate table.

SQO's Normalization

For reading purpose, the Where operator is used in the examples, but in this current phase of the normalization of the expression tree it doesn't exist anymore because it's substituted by its normalized form explained in section 3.3.16.

So the normalized branch that will be inserted in the tree is:

- One-to-Many:

$$e_1.FILTER(e_1.Select(q \mapsto e_2));$$

where e_1 represents the external table and e_2 the join condition.

- One-to-One:

$$e_1.FILTER(e_1.Select(q \mapsto e_2)).Single();$$

where e_1 represents the external table and e_2 the join condition.

3.5 (Un)Boxing the tree

The LINQ's data model allows the developer that writes the query to shape as he wants the result. This leads to complex record structures that can also have an arbitrary level of nesting of tuples and lists inside the intermediate result and in the result of a query.

For example, the result type of this query will be an object containing the name of a customer and a list of his orders:

```
1   var query = db.CUSTOMER.Select(v => new { name = v.C_NAME, orders = v.
       ORDERS });
```

The result will consists of two nested lists:

[(name = x_0, orders = [$y_0, \ldots, y_m$]), ..., (name = x_n, orders = [$y_0, \ldots, y_k$])]

Since the generated SQL code has to run on a database back end, and a pure relational database only has rows and tables as structures, those LINQ complex structures has to be mapped back to them or recreated using a combination of them.

In LINQ, as structures, we can have:

- Records: $(x_1, \ldots, x_n) ::= t$
- Atomic elements: $y ::= int, String, float, \ldots;$
- Lists: $[t]$

As said, in a pure relational database, we only have:

- Rows: $(a_1, \ldots, a_n) ::= r$;
- Tables: $[r]$.

The mapping can be easily summarized as: if it's a collection where operators can iterate over, then it maps to a relational *table*, when not it maps to a simple *row*.

Therefore those mappings from LINQ to database structures are:

- Record $\Mapsto$ Row;
- Atomic element $\Mapsto$ Row;

- List ⤇ Table.

A more complex case is when lists are nested. That can't be handled directly but needs to be normalized first.

Those nesting can be handled by emitting a different relational query for each nesting level and link them using surrogate values [3]. Each query collapses all the list of its nesting level inside one single table, with a new attribute inside which the surrogate value is specified to maintain the original binding with the outer table.

In our example two relational queries will be emitted:

1. Q_0 that represents the relation encoding of the outer list with surrogate values replacing the inner, even empty, lists;
2. Q_1 that represents all the inner lists in just one single table with a new surrogate key column to link them back.

This method will generate a record for the outer list, that as previously shown can be simply represented with a *row*, and a list for the inner ones, that can represented using a *table*, as shown in figure 3.3:

- [[t]] and (. . . , [t], . . .) ⤇ row + table;

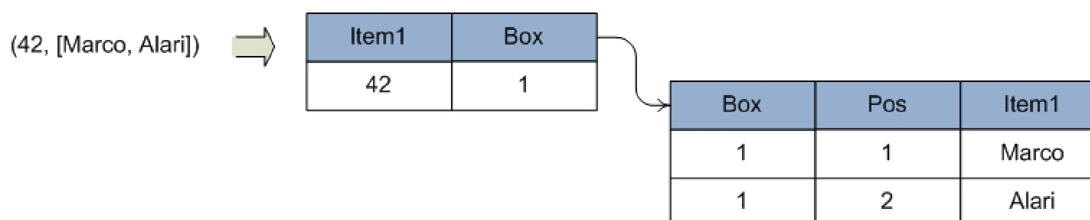


Figure 3.3: Application of the Box operator

Each operator in LINQ expects the result of its subtrees to be of a specific type: for example a Select operator expects its source to be a collection of elements, therefore a *table*, and its lambda expression to return a record, therefore a *row*. The provider must, through invoking the built-in operators Box and UnBox [9], ensure that the subtrees are of the right type.

The Box Operator

The Box operator forces the forking of the query compilation by inserting the surrogate values substituting the nested lists, and therefore generating the needed relational queries.

It's applied when a subtree *e* is of type *table* but its parent node expects it to be of type *row*.

Looking again at our example, a Box operator needs to be placed on the *order* list, because a parameter of a record expects its subtree to be always of type *row* and not of type *table*.

The UnBox Operator

The UnBox operator, contrary to the Box operator, forces the deference of those surrogate values, by introducing a Join based on them to merge the various nesting tables.

It's applied when a subtree *e* is of type *row* but its parent node expects it to be of type *table*.

For example here:

```
1  var query = db.CUSTOMER.Select(v => new { name = v.C_NAME, orders = v.
       ORDERS }).Where(h => h.orders.Count() > 0);
```

As before, a box operator is applied to let the nested order list fit into the parameter *orders* which expects a row, but then, when the operator Count is called on that parameter, the UnBox operator is applied because Count works on lists (therefore expects a table) but now *order* is boxed, and so it's of type *row*.

3.5.1 Introduction rules

To ensure that the types matches inside the tree, it has to be visited again. During this phase the Box and UnBox operators must be inserted where needed.

For each type of node of our AST the expected type of its eventual subtree and its return type has to be map to either *row* or *table* using the mapping rules presented above. Basically the subtree needs to be of type *table* when it will be iterated over by some operator and in all the other cases it will be of type *row*. Knowing that it's intuitive to know what's the expected or returned type of a subtree. The function call however has their own prototype where those types are specified.

Here are presented a set of rules that ensures that there will be no type mismatch in the tree, so that also after that mapping all will match again. This approach is compositional, and no environment is needed as long as inside each node of our tree is specified its return type, so a node can check directly the type of its kids. That can be specified when the node is created and can

be hard coded inside it because it can't be changed during the normalization process. The Box and UnBox operator has to be seen as cast operators, so they will not modify the type of a node.

In the rules the operator $\Mapsto$ means *compiled into*. The result of the compilation is a structure containing the original subtree with the introduction of the Box and UnBox operators.

For example:

$$e_1 \;\Mapsto\; e'_1$$

means that the subtree e_1 is compiled into a new subtree e'_1 on which the normalization process is already applied.

Table

A Table node it of type *table* because functions can iterate over it.

$$\frac{}{Table(c_1,\ \ldots,\ c_n) \;\Mapsto\; Table(c_1,\ \ldots,\ c_n) \;::\; tbl}$$

Record

In the record rule, each parameter expects it's subtree to be a value, therefore of type *row*. The rule ensures that by forcing the introduction of the Box operator in case the subtree of the parameter is of type *table*. Its return type is *row* because it instantiate a new occurrence of a record.

$$\frac{e_i \;\Mapsto\; e'_i \;:\; \tau_i \,|_{i=1,\ \ldots,\ n}}{new\left\{\begin{matrix} f_1 & = & e'_1 \\ & \vdots & \\ f_n & = & e'_n \end{matrix}\right\} \;\Mapsto\; new\left\{\begin{matrix} f_1 & = & \boxed{\downarrow}^{\tau_1}_{row}(e'_1) \\ & \vdots & \\ f_n & = & \boxed{\downarrow}^{\tau_n}_{row}(e'_n) \end{matrix}\right\} \;:\; row}$$

Member Access

A member can be accessed only on a tuple or a record, therefore the subtree must be of type *row*. Its return type is *row*.

$$\frac{e \;\Mapsto\; e' \;:\; row}{e.n \;\Mapsto\; e'.n \;:\; row}$$

Variable

A variable represents a single value, therefore its return type is *row*.

$$\frac{}{v \;\Mapsto\; v \;:\; row}$$

Constant

A Constant represents a single constant value, therefore its return type is *row*.

$$\frac{}{c \;\Mapsto\; c \;:\; row}$$

Binary

A binary operator is applied to a pair of single values, therefore both its operand must be of type *row*. Its return type is *row* because it returns the result, a single value.

$$\frac{e_i \;\Mapsto\; e'_i \;:\; row\,|_{i=1,\,2}}{e_1 \circledast e_2 \;\Mapsto\; e'_1 \circledast e'_2 \;:\; row}$$

Unary

A unary operator is applied to a single value, therefore its operand must be of type *row*. Its return type is *row* because it returns the result of the operation, namely a single value.

$$\frac{e_i \;\Mapsto\; e'_i \;:\; row \,|_{i=1,\,2}}{\circledast(e_1) \;\Mapsto\; \circledast(e_1) \;:\; row}$$

Conditional

In a conditional expression the type of the cases must be the same. Therefore the false statement must match the true one. This is done by forcing one to be of the same type as the other. Its return type is the same as the statements.

$$\frac{e_i \;\Mapsto\; e'_i \;:\; \tau_i \,|_{i=1,\,\ldots,\,3}}{e_1 \,?\, e_2 \;:\; e_3 \;\Mapsto\; e'_1 \,?\, e'_2 \;:\; \boxed{\Downarrow}^{\tau_3}_{\tau_2}(e'_3) \;:\; \tau_2}$$

Lambda

The subtree representing the body of the lambda expression has no restriction on which type it has to be. The return of a lambda operator must be however of type *row* because a lambda expression returns always a single tuple.

$$\frac{e \;\Mapsto\; e' \;:\; \tau}{v \;\Mapsto\; e \;\Mapsto\; v \;\Mapsto\; \boxed{\Downarrow}^{\tau}_{row}(e') \;:\; row}$$

Function Call

A function call has its own defined prototype where, for each parameter, the expected type is specified. This rules ensures that the given subtree for each of them matches the expected one. The return type of a function call its also specified by the prototype but no cast operation are needed because it depends by the implementation of the function itself.

$$\frac{e_i \;\Mapsto\; e'_i \;:\; \tau_i \,|_{i=1,\,\dots,\,n} \qquad f \;::\; (t_1,\;\dots,\;t_n) \;\mapsto\; t_{n+1} \qquad \sigma_i \;=\; \begin{cases} tbl & t_i = [t] \\ row & otherwise \end{cases}}{e_1.f(e_2,\;\dots,\;e_n) \;\Mapsto\; \Downarrow^{\tau_1}_{\sigma_1}(e'_1).f(\Downarrow^{\tau_2}_{\sigma_2}(e'_2),\;\dots,\;\Downarrow^{\tau_n}_{\sigma_n}(e'_n)) \;:\; t_{n+1}}$$

Select

The Select function call expects a source of type *table* because it iterates through a collection. The lambda expression must be of type *row* because it consumes a tuple at each iteration. Its return type is *table* because a select returns again a collection.

$$\frac{e_i \;\Mapsto\; e'_i \;:\; row \,|_{i=1,\,2}}{e_1.Select(v \;\Mapsto\; e_2) \;\Mapsto\; \Downarrow^{\tau_1}_{table}(e'_1).Select(v \;\Mapsto\; \Downarrow^{\tau_2}_{row}(e'_2)) \;:\; tbl}$$

3.6 The Running Example

3.6.1 The LINQ Query

As running example, the following query is used:

```
1  var query = db.CUSTOMER.Select(v => new
2      {
3        name = v.C_NAME,
4        region = v.NATION.REGION.R_NAME,
5        orders = v.ORDERS.
6            GroupBy(key => key.O_ORDERSTATUS, (key, values) =>
7              new {
8                status = key,
9                info = values.Select(x => new { price = x.O_TOTALPRICE, date =
                       x.O_ORDERDATE }),
10               num = values.Select(x => x.LINEITEM.Count())
11             })
12     });
```

It returns for each customer an object containing its name, region and orders, grouped by their status. Each grouping key is bounded to an object containing informations (status, price, order date and the number of items) about the orders inside the group.

3.6.2 The LINQ expression tree

In figure 3.4 a graph representing the expression tree generated at compile time by the C# compiler is shown.

3.6.3 The SQO-Normalized tree

In figure 3.5 is shown the tree resulting after the normalization of the SQO explained in 3.3.

As we can see, the GroupBy is substituted by its normalized version shown in sqo:groupby: *_LINQ2SQL_0* is the new generated variable that represents a whole tuple. It is used, as explained, to recreate the object the user wants to be bounded to the keys.

This tree uses the new defined light-weighted nodes shown in section 3.2.

3.6.4 The Un-virtualized tree

As next step the virtual attribute are made explicit, as explained in 3.4. In figure 3.6 the new resulting tree is shown.

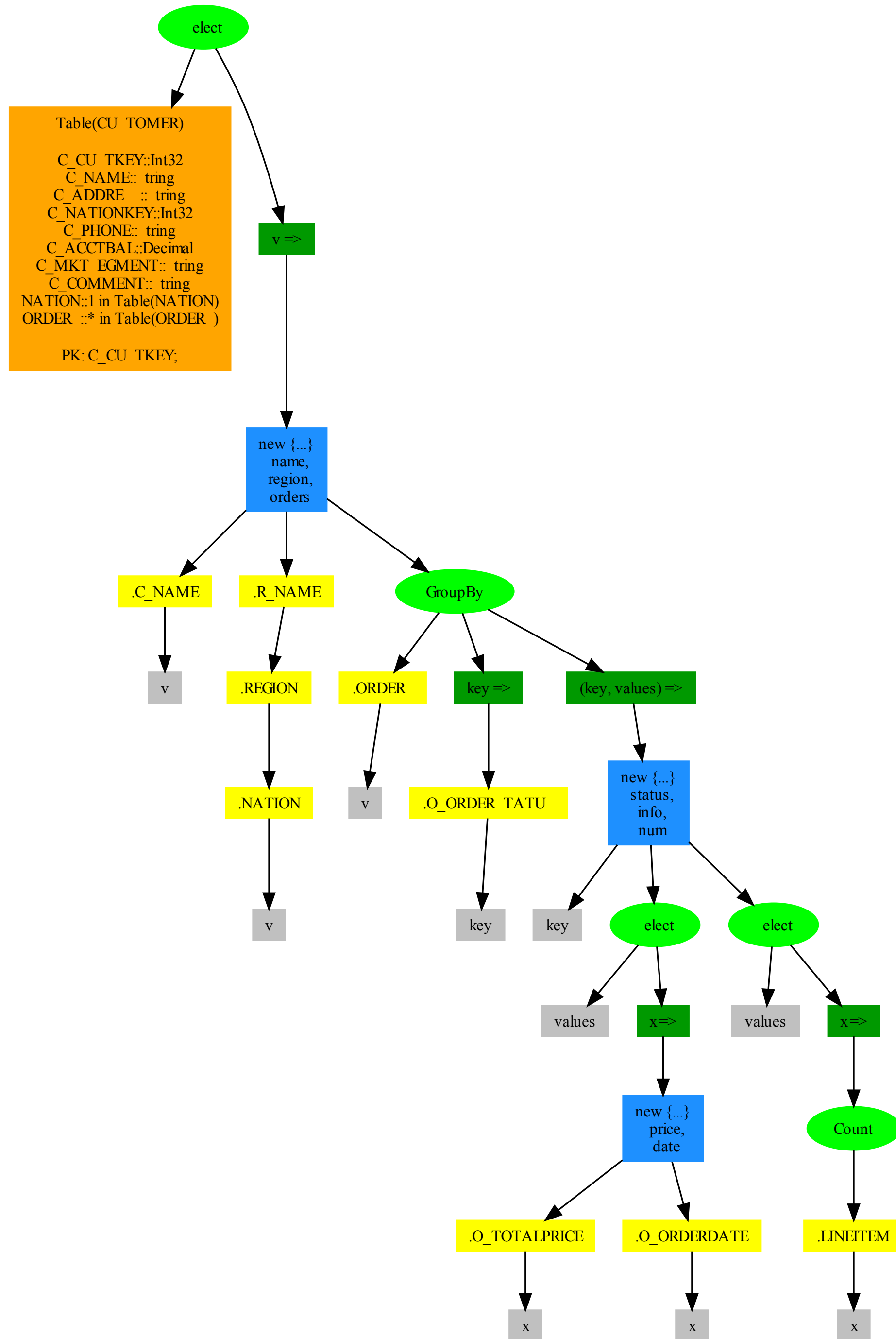


Figure 3.4: The LINQ expression tree for the query in 3.6.1

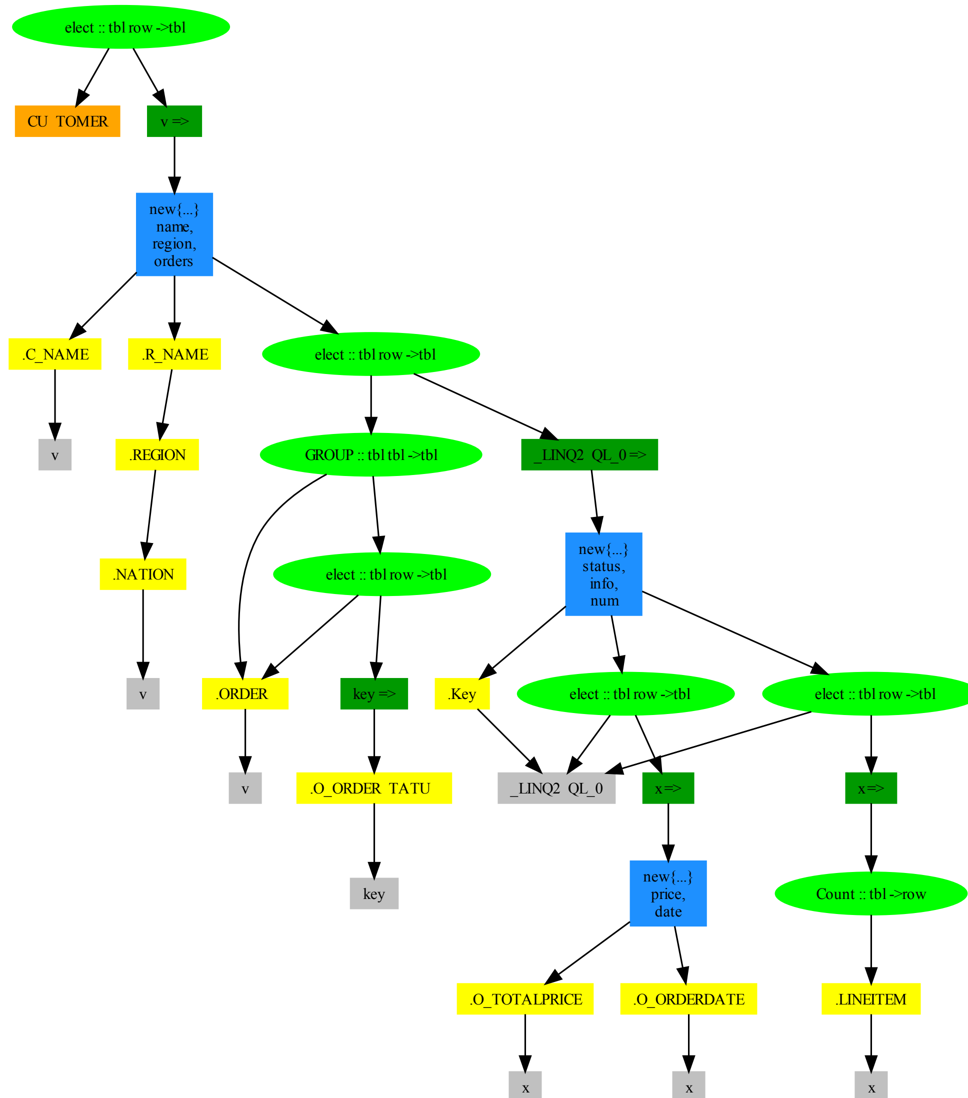


Figure 3.5: The SQO-normalized expression tree for the query in 3.6.1

Various virtual attributes are accessed in the query:

- *Nation*, of the table *Customer*, which is used to navigate to the table *Region* through its virtual attribute *Region*;
- *Orders*, of the table *Customer* ;
- *LineItem*, of the table *Orders*.

According to the rules, the relative join branches based on the *foreign keys* are inserted, in order to merge the tables and make them explicit. The other virtual attributes inside the tables are not un-virtualized because they are not accessed

3.6.5 The (Un)Boxed tree

The last step of the normalization is the introduction of the Box and UnBox operator to simplify the object world's complex structure in order to be executed on a pure relational DBMS. In figure 3.7 the tree after the application of the rules shown in section 3.5 is presented.

As we can see the Box operator is used when a tuple is expected, and the UnBox operator is introduced when something to iterate over is expected.

For example, the anonymous type definition at the top of the tree, expects all its parameter to be tuples, but the *order* parameter is a list of orders. Therefore the Box operator is applied on the top of the subtree representing it.

On the other hand, the UnBox operator is applied to the new generated variable *_LINQ2SQL_0* because the Select operator expects as first parameter a list of elements on which it can iterate. The Select's result is then again boxed to fit into the *info* parameter of the anonymous type.

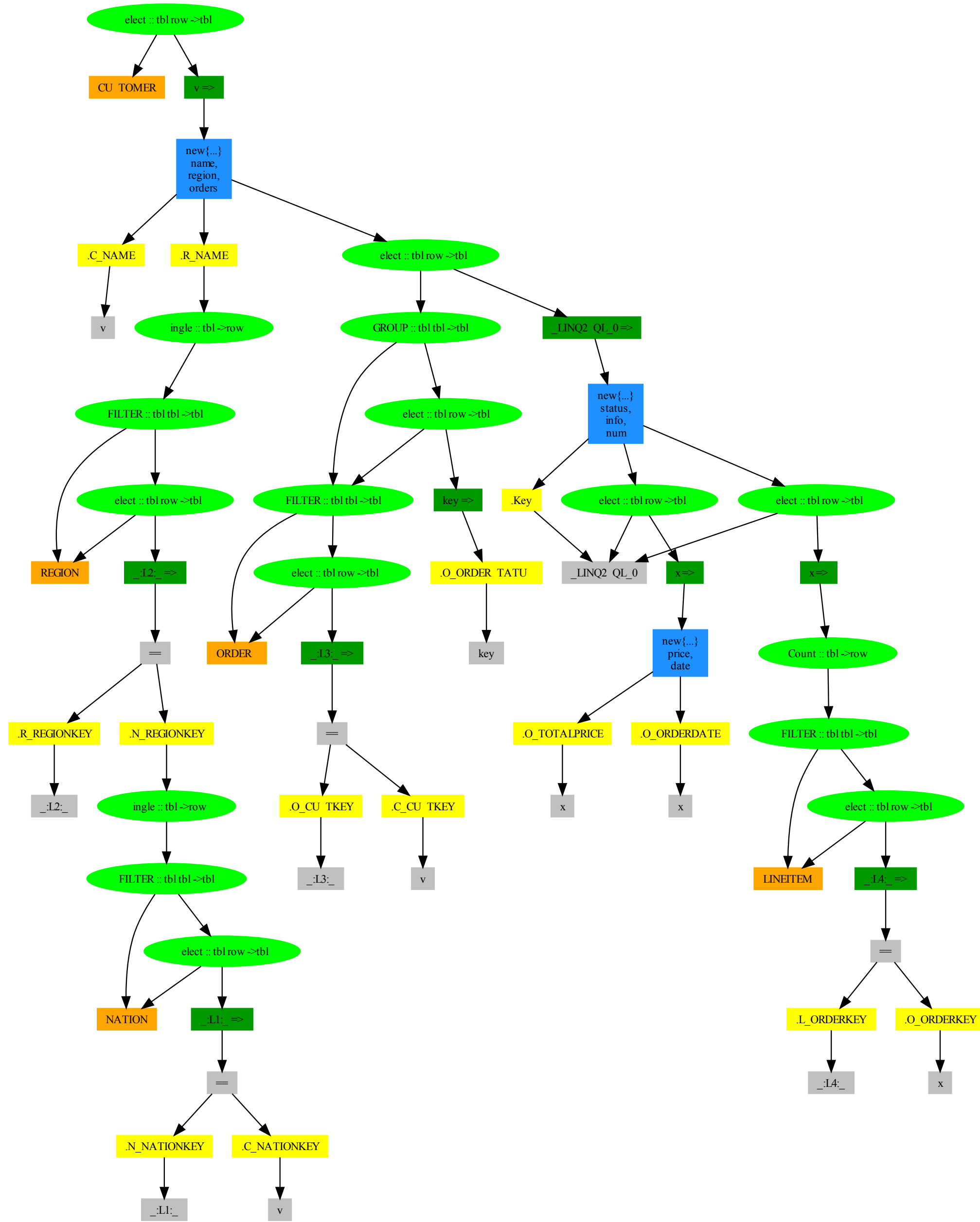


Figure 3.6: The un-virtualized tree for the query in 3.6.1

Figure 3.7: The (un)boxed tree for the query in 3.6.1

Chapter 4

Algebraic compilation

The last step before generating SQL code is to compile our normalized tree in an intermediate algebraic query bundle, where each query represents a level of nesting of the query's result. That's done so the Pathfinder optimizer can then be used to obtain a heavily optimized plan and a new level of abstraction is added because this new algebraic query bundle can be compiled into different output codes. In our case we will use the SQL code generator offered Pathfinder to obtain SQL code that will run on every SQL:1999 capable DBMS.

During this process out of our normalized AST a new directed acyclic graph, where each node encodes a logical algebra operator, will be created using the rules described in section 4.4. This new tree will be composed of operators defined in a algebra shown in section 4.1, whose operators match the capabilities of modern query processors.

The *Loop lifting* technique is used to fully exploit the *set-oriented evaluation* typical of DBMS, where, in absence of dependencies between rows, the system can process them in any order. This technique realizes this needed independence of the iterated evaluations by compiling them into a single table containing the encodings for all the iterations instead of performing n independent evaluation of the subtree e.

4.1 The Pathfinder Algebra

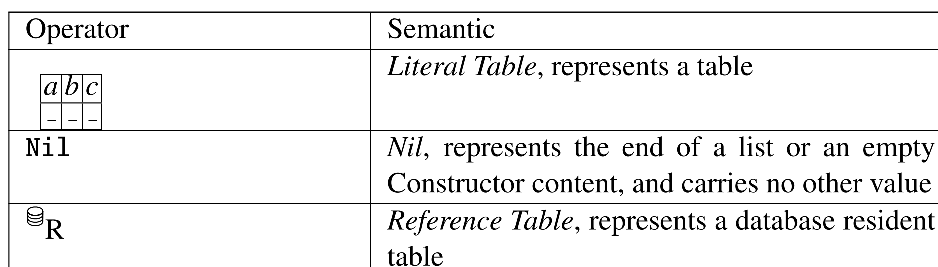
The target language of the compiler is a relational algebra, whose operators match the capabilities of modern query processors, that can be understood by the Pathfinder engine for optimization and translation into SQL code. The actual Pathfinder algebra can be seen more in

details here: http://wiki.pathfinder-xquery.org/wiki/index.php/Logical_Algebra.

Having an intermediate algebraic language introduces a new level of abstraction between the data and the query, allowing the same plan to be translated into different languages by only having a new translator. It also allow us to use the Pathfinder query optimizer and its SQL code generator to get our wished result.

Table 3.3 lists the available operators, which will form the node of our new DAG.

Operator	Semantic
	Binary operators
$R_1 \times R_2$	*Cross*, Cartesian product between the relations
$R_1 \setminus R_2$	*Difference*, difference between the relations
$R_1 \dot{\cup} R_2$	*Disjunion*, UNION ALL between the relations
$R_1 \bowtie_{<a_i:b_i>} R_2$	*Equi Join*, equality join between the relations based on the column a_i and b_i
$R_1 \cap R_2$	*Intersect*, returns the common elements of the relations
$\oplus$(R)	*Serialize*, root of the algebraic plan
$R_1 \bowtie_{p_1, \ldots, p_n} R_2$	*Theta Join*, join between the relations based on $p_1, \ldots, p_n$ predicates
	Unary operators
$\texttt{agg}_{res:<col>/c}(Rel)$	*Aggregates*, groups row by c, then attaches aggregated *col* in *res*
$@_{a:v}(Rel)$	*Attach*, attaches a new column a, whose value is the constant v
$\circledast_{res:<b_1, \ldots, b_n>}(Rel)$	*Operators*, calculates its result and stores it in the new column *res*
$\texttt{cast}_{a:(t)b}(Rel)$	*Cast*, creates a new column a which contains the values of b casted to the type t
$\texttt{Count}_{res/c}(Rel)$	*Count*, groups the rows by column c and then counts the tuples, saving the result in column *res*
$\delta(Rel)$	*Distinct*, eliminates duplicates
$\pi_{a_1:b_1, \ldots, a_n:b_n}(Rel)$	*Projection*, projects out columns b_i (optionally renamed into a_i)
$\texttt{rank}_{res:<b_1, \ldots, b_n>}(Rel)$	*Rank*, attaches an arbitrary ranking in column *res*, based on the order $< b_1, \ldots, b_n >$
$\texttt{Rowid}_{res}(Rel)$	*Rowid*, attaches a column *res* filled with unique values
$\#_{res:<b_1, \ldots, b_n>/c}(Rel)$	*RowNum*, attaches a dense ranking in column *res*, based on the order $< b_1, \ldots, b_n >$ and partitioned by c
$\varrho_{res:<b_1, \ldots, b_n>}(Rel)$	*RowRank*, attaches a dense ranking in column *res*, based on the order $< b_1, \ldots, b_n >$
$\sigma_p(Rel)$	*Select*, select tuples that satisfies the condition p
	Leaf
a b c / – – –	*Empty Table*, represents a table without any tuples



Operator	Semantic
$\begin{array}{\|c\|c\|c\|}\hline a & b & c \\ \hline - & - & - \\ \hline \end{array}$	*Literal Table*, represents a table
`Nil`	*Nil*, represents the end of a list or an empty Constructor content, and carries no other value
⛁$_{\mathrm{R}}$	*Reference Table*, represents a database resident table

Table 4.1: The *Pathfinder* algebra operators

$$\begin{array}{rl} \circledast & (\in \{+,\ -,\ *,\ <,\ >,\ \mathit{and},\ \mathit{or}, \dots\}) \\ \texttt{agg} & (\in \{\mathit{avg},\ \mathit{max},\ \mathit{min}, \dots\}) \end{array}$$

On the implementation side, each of this operators must be modeled by its own class where its semantical informations are stored. When the compilation rules showed in 4.4 are applied to our tree, they create out of it a new DAG composed with objects, whose type is among those new defined classes that represents the Pathfinder algebra operators. The class diagram is shown in figure 4.1.

4.1.1 Binary Operators

The Binary Operator super-class contains the general structure of the binary operators. Each binary operator works on two relations, whose relative subtree has to be stored.

Cross

The cross product operator creates the Cartesian product of two relations. Their column names must be distinct.

No further information needs to be saved.

Difference

The difference operator returns the difference between two relations which have the same structure. The result are all tuples of the left child that have no corresponding tuple in the right child.

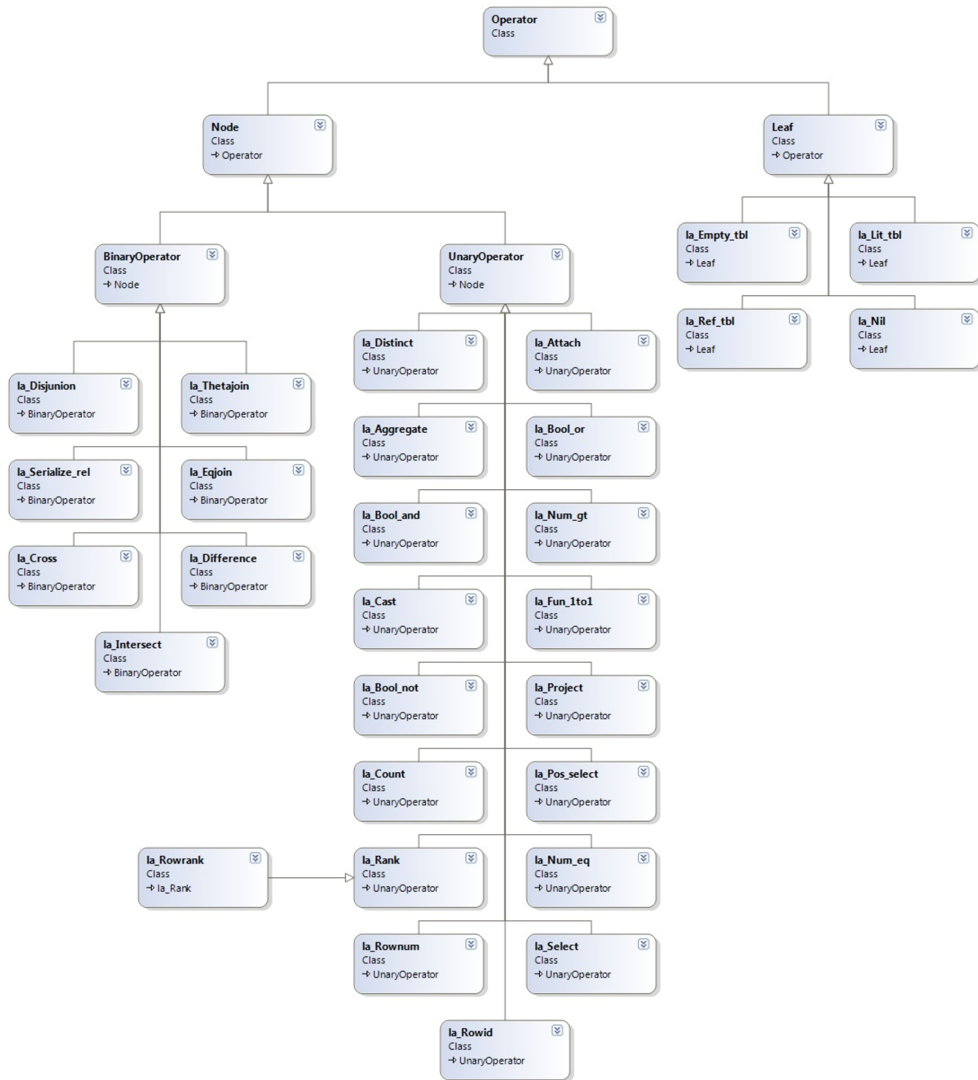


Figure 4.1: The algebra class diagram

No further information needs to be saved.

Disjunion

The Disjunion operators represents the UNION ALL operator in SQL. It combines two relations without removing duplicates.

No further information needs to be saved.

Equi Join

The equi-join operator joins two input relations based on a single equality condition. As for the cross product column names are ensured to be distinct for both inputs.

The name of the join key columns are stored inside the node.

Intersect

The intersect operator works on two relations with identical column names. Opposed to the Difference operator, it returns the common elements of the two relations.

No further information needs to be saved.

Serialize Relation

The table serialize operator is the root of a logical algebra DAG and represents a result table. The left child references the side effects (traces and runtime error checks) and the right child contains the DAG that represents the resulting table. The right child is expected to provide at least three columns (the names of these columns are stored in the semantical field of the serialize operator (schema)) that represents the iteration (partitionBy), position (orderBy), and

item values (columnNames).

The information saved in the node is the table schema with the column names. Due to optimizations, the columns iteration and position, would probably reference an item column. Therefore also those references are to be saved.

ThetaJoin

The theta-join operator joins two input relations based on multiple predicate conditions. As for the cross product all column names are ensured to be distinct.

The information saved in the node is an array representing the conditions (for each condition the two join key columns and the comparison operand, whose name inside the algebra can be seen in table 4.4, are specified).

4.1.2 Unary Operators

The Unary Operator super-class contains the general structure of the unary operators. Each unary operator works on one relation, whose subtree has to be stored.

Aggregate

The aggregate operator applies the operator to the input column and saves the result in a new column, which will be the only one in the resulting table. If a partitioning attribute is specified, aggregates are calculated for each input group partitioned by that attribute.

The information saved in the node are the type of aggregate (whose name inside the algebra can be seen in table 4.3), the name of the result column and the eventual partition attribute.

Attach operator

The attach operator extends the input relation with a new column, whose values is the constant v.

The information saved in the node are the name of the new column and its value.

Bool and

The boolean and operator works on tuple by tuple basis and extend the input schema with a new boolean column containing the result of the logical operator *and* between the two specified columns.

The information saved in the node are the name of the resulting column and the name of the columns on which the operator is applied.

Bool not

The boolean not operator works on tuple by tuple basis and extends the input schema with a new boolean column containing the result of the logical operator *not* on a specified columns.

The information saved in the node are the name of the resulting column and the name of the column on which the operator is applied.

Bool or

The boolean or operator works on tuple by tuple basis and extends the input schema with a new boolean column containing the result of the logical operator *or* between the two specified columns.

The information saved in the node are the name of the resulting column and the name of the columns on which the operator is applied.

Cast

This operator works on tuple by tuple basis and extends the input schema with a new column with a specified type. This new column is based on the values inside a column which is the input to a cast. The input column can have a polymorphic type. If a cast fails an error should be triggered.

The information saved in the node are the name of the input and resulting columns and the type at which it will be casted.

Count

The count operator counts the tuples of an input relation and saves the result in a new column, which will be the only one in the resulting table. If a partitioning attribute is specified, the count is calculated for each input group partitioned by that attribute.

The information saved in the node are the name of the result column and the eventual partition attribute.

Distinct

The distinct operator removes from a relation all the duplicate tuples.

No further informations needs to be saved inside the node.

Function

This class represents calls to generic mathematical functions. In table 4.2 a list of supported function can be found along with the related name inside the algebra. The result is saved in a new column.

The information saved in the node are the names of the columns on which the function is applied and of the resulting column and the function applied.

Equality - Greater Than

The equality operator works on tuple by tuple basis and extends the input schema with a new boolean column containing the result of the operator between the two specified columns. This two operators can be combined to obtain other comparison operators such $<$, $\geq$ or $\geq$.

The information saved in the node are the name of the resulting column and the name of the columns on which the operator is applied.

Project

The project operator prunes and renames columns of the input relation.

The information saved in the node are the name of the columns that will be projected out at their eventual new names.

Rank

The rank operator creates a new column that contains arbitrary ranks in the order given by a list of sort columns. The operator has no partition column.

The information saved in the node are the name of the ranking and of the resulting columns.

Rowid

The rowid operator provides a new column that is filled with unique values. The operator has no partition column.

The information saved in the node are the name of the ranking and of the resulting columns.

RowNum

The rownum operator creates a new column that contains enumeration values starting from the value 1. Like the aggregates the rownum operator may have a partitioning column that ensures that each partition starts the enumeration from the value 1. The order that is used to assign the enumeration values is based on a list of sort columns.

The information saved in the node are the name of the sorting, of the resulting and of the eventual partition columns.

RowRank

The rowrank operator creates a new column that contains dense ranks (starting from the value 1) for each group found in the order given by a list of sort columns. The operator has no partition column.

The information saved in the node are the name of the sorting and of the resulting columns.

Select

The select operator throws away all the tuples whose value in the selection column is false. It works only on boolean columns.

The information saved in the node is the name of the selection column.

4.1.3 Leaf Nodes

The Leaf Nodes super-class contains the general structure of the leaf nodes. Since leaf nodes have no children, no information is stored in it. It's only used for readability.

Literal table

The literal table represents a new literal base table. In most cases the literal table consists of a single column table with a single tuple inside.

The information that needs to be saved inside the node are the columns (their names and types) and the stored values.

Empty table

The empty table is a literal table that contains no tuples.

The information that needs to be saved inside the node are the column names.

Nil

The nil operator represents the end of a list or an empty constructor content, and carries no other value. It can be safely ignored.

No information needs to be saved inside the node.

Reference table

The literal reference table represents a database resident table.

The information that needs to be saved inside the node are the name of the table, the columns (their names and types), their new names and the stored values.

4.2 Types mappings

To let the operands match the ones defined inside the Pathfinder algebra dialect, they need to be mapped before create a node of the algebraic plan.

In this section all the necessary (until now) mappings are presented.

Name	Algebra name
+	add
−	subtract
/	divide
%	modulo
*	multiply

Table 4.2: Functions 1 to 1

Name	Algebra name
Average	avg
Max	max
Min	min
Sum	sum

Table 4.3: Aggregators

Name	Algebra name
==	eq
$>$	gt
$\geq$	ge
$<$	lt
$\leq$	le
! =	ne

Table 4.4: Comparison operators

4.3 Loop-lifting

The *loop-lifting* approach [5, 4] permits the evaluation of the loop body e of a Select operator for all its iterations in parallel.

Normally the body is evaluated for each binding making it *data driven*: the more data it's queried, the more evaluation are performed. Those evaluation are among them independent, therefore conceptually is possible to execute them at the same time in parallel. The idea is to replace all free occurrences of the variable v for all bindings in the body e with the value it will have at that iteration and then evaluate the body, instead of evaluating the body for each binding.

The result of this procedure is a single table containing all the bindings of the body, with a *iter* column, which distinguish separate logical iterations, and a *pos* column, which represents the position of the element inside the iteration.

In our compilation rules this technique is adapted to work with typical structure of the object world, like records and nested lists.

4.4 Compilation Rules

In this section the compilation rules are shown. Applying this rules to the normalized expression tree will produce the algebraic DAG. The rules are compositional meaning that can be applied in each order ensuring always the right result.

The result of each rule is a struct that represents the algebraic representation of the given tree. The relational resulting table must have a fixed structure with at least three columns: *iter*, *pos*, and an arbitrary number of *item* columns. The *iter* column contains the reference to the iteration, the *pos* column the absolute position of the tuple inside the iteration and the *item* columns the actual data.

In the rules the operator $\Rrightarrow$ means *compiled into* in the context of the given *loop* relation and the Γ environment. The result of the compilation is the algebraic plan in the structure described in section 4.4.1.

The *loop* relation contains the number of independent iterations that has to be performed. The Γ environment keeps track of the algebraic tree representing the value of the variables.
At the beginning, the *loop* relation is a singleton relation, whose value is set to 1 meaning that

only one iteration has to be performed. The Γ environment is set to null because it will be fed during the compilation when a variable is encountered.

Each compilation rule passes the *loop* relation and the Γ environment top-down in order to compile its subtrees, while the resulting algebraic DAG is generated bottom-up.

For example:

$$\Gamma;\ loop\ \vdash e\ \Mapsto\ (q_e,\ cs_e,\ ts_e)$$

means that the subtree e is compiled in the context of the *loop* relation and the Γ into a new structure containing its algebraic plan, its relational structure and the eventual plans for the nested lists of the outer query.

For ease of readability, real names are used when columns are renamed or introduced (as, for example, the result of a binary operator). Pathfinder doesn't understand such names, it only handles the names *iter*, *pos* and *item* that can have as postfix a number, so when the rules are implemented those name must be substituted by names that Pathfinder can handle.

4.4.1 The *QCsTs* structure

A compiled tree is described by a structure composed of three elements:

- *q*: represents the algebraic tree;
- *cs*: contains the schema of the resulting relation;
- *ts*: contains eventual surrogate tables generated to handle nested lists.

Those three information combined together form the algebraic plan of a query.

q

The *q* entry contains the algebraic tree.

cs

The *cs* entry contains a representation of an object's schema where its variables maps to columns of a relational table. This object is the result of the algebraic plan q.

In Pathfinder there are no complex or nested structure, just flat, because all has to be in the first normal form. Therefore each object must be represented in flat form as a relational table, and that means that each property of the object must map to a column of a relational table. Those mapping between object attributes and relational columns are stored inside this structure.

Due to Pathfinder allowing only column names in the form of the string *item* with a numerical postfix, to make shift operations easier, inside the *cs* structure the variables map to a number instead of the complete name. The name of the column is generated when the information are needed, by concatenating the index with the string *item*.

The mapping must be positional and must maintain the original nesting, meaning that a record structure like:

```
1  new { a = 20, b = new {c = 10, d = 40} }
```

should be mapped like:

$\{a \mapsto 1, b \mapsto \{c \mapsto 2, d \mapsto 3\}\}$

A position must be assigned incrementally only to properties that have an atomic value that can be stored directly into a column of a relational table. When that doesn't happen, like for the *b* variable in our example that maps to a complex structure, its internal structure must be mapped into an another *cs* structure that will be mapped by the variable name.

A visual example of the map of an object to a relational table is shown in figure 4.2.

The *cs* structure must therefore allow a variable to map to either a position or a new *cs* struct. The variable name, in same cases, won't be specified, so just a position is specified, like when a table is mapped into the *cs*. In this case the column name maps to its position in the table's schema.

ts

The *ts* entry contains the eventual surrogate tables that represents the nested lists that are present in the result. As explained in section 3.5, nested lists will be substituted in the outer query

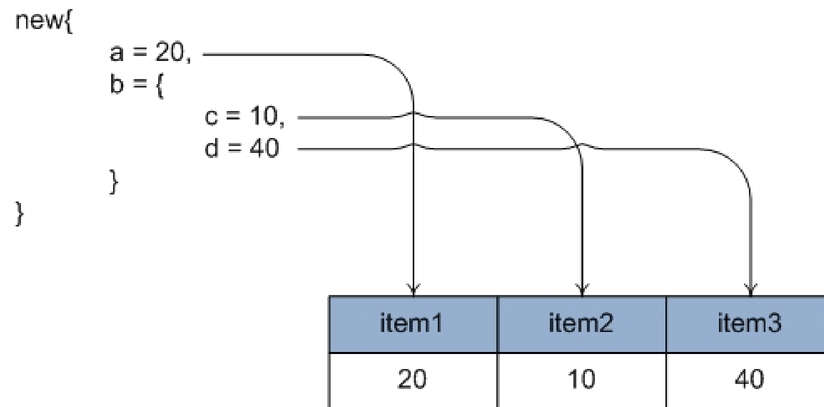


Figure 4.2: A mapping of an object to a relational table

with surrogate values and a new query, representing a table containing all inner lists, will be generated. Inside the *ts* structure is present an entry for each surrogate column along with the corresponding query for the surrogate table. The query will be specified in this same *QCsTs* structure because it's a relational query again and can therefore have other nesting levels.

4.4.2 Helper functions and operators

Some helper functions are used inside the compilation rules to handle the informations inside the *cs* and *ts* structure.

Leafs

This function returns the leafs nodes of a *cs* entry.

$$cs \equiv \{a \mapsto 1, b \mapsto \{c \mapsto 2, d \mapsto 3\}\}$$

$$Leafs(cs) \equiv [1, 2, 3]$$

Shift Leafs

This function returns the given *cs* or *ts* with positions shifted forward of an offset *i*.

$$cs \equiv \{a \mapsto 1, b \mapsto \{c \mapsto 2, d \mapsto 3\}\}$$

$Shift(cs,\ 1)\ \equiv\ [2,\ 3,\ 4]$

The function *Decr* shifts the positions backwards.

Offset

This function returns the number of leafs of a *cs* structure.

$cs\ \equiv\ \{a\ \mapsto\ 1,\ b\ \mapsto\ \{c\ \mapsto\ 2,\ d\ \mapsto\ 3\}\}$

$Offset(cs)\ \equiv\ 3$

cardBefore

This function returns the number of leafs of a *cs* structure before a member, even if nested.

$cs\ \equiv\ \{a\ \mapsto\ 1,\ b\ \mapsto\ \{c\ \mapsto\ 2,\ d\ \mapsto\ 3\}\}$

$cardBefore(cs, d)\ \equiv\ 2$

Keys

The function *Keys* returns which columns of the *cs* structure are surrogates column.

The inverse function *notKeys* returns which columns of the *cs* structure are not surrogates column.

$cs\ \equiv\ \{a\ \mapsto\ 1,\ b\ \mapsto\ 2\}$
$ts\ \equiv\ \{1\ \mapsto\ \Box_1\}$

$Keys(cs)\ \equiv\ [1]$

retrByKeys

The function *retrByKeys* returns a *ts* structure with only the key appearing in a given list.

$$ts \equiv \{1 \mapsto \square_1,\ 3 \mapsto \square_3,\ 5 \mapsto \square_5\}$$

$$retrByKeys(ts,\ [1,5]) \equiv \{1 \mapsto \square_1,\ 5 \mapsto \square_5\}$$

The Concatenate operator

The || operator append a *cs* or a *ts* structure at the end of an existing one. It also shifts the indexes of the appended structure to ensure unique indexes.

$$cs_1 \equiv \{a \mapsto 1,\ b \mapsto \{c \mapsto 2,\ d \mapsto 3\}\}$$

$$cs_2 \equiv \{e \mapsto 1,\ f \mapsto 2\}$$

$$cs_1 \,||\, cs_2 \equiv \{a \mapsto 1,\ b \mapsto \{c \mapsto 2,\ d \mapsto 3\},\ e \mapsto 4,\ f \mapsto 5\}$$

The Append operator

The $\oplus$ operator [9] appends recursively the *ts* structure of two relations, based on a given q algebraic tree.

The operator represents the rule:

$$\{\ldots,\ c \mapsto (q_1,\ cs,\ \{\ldots,\ c_i \mapsto \square_{i,1},\ldots\})\} \overset{q_0}{\oplus} \{\ldots,\ c \mapsto (q_2,\ cs,\ \{\ldots,\ c_i \mapsto \square_{i,2},\ldots\})\} =$$

$$\Big\{\ldots,\ c \mapsto \Big(\pi_{iter:up,pos,c_i:down}\Big(\pi_{lr,iter:c,up:down}(q_0) \bowtie_{lr,iter} \overbrace{(\varrho_{down:<iter,lr,pos>}(@_{lr:1}(q_1)\ \dot{\cup}\ @_{lr:2}(q_2)))}^{q_r}\Big),$$

$$cs,\ \{\ldots,\ c_i \mapsto \square_{i,1},\ldots\})\} \overset{q_r}{\oplus} \{\ldots,\ c \mapsto (q_2,\ cs,\ \{\ldots,\ c_i \mapsto \square_{i,2},\ldots\}\Big)\Big\}$$

The Slicing operator

The $\overset{q}{\lhd}$ operator [9] spreads a filtering operation down the *ts* structure in order to eliminate eventual nested lists whose outer surrogate key was filtered out. It's however not mandatory to apply it. It can be used to have more polished data to handle.

The operator represents the rule:

$$\overset{q_0}{\lhd}(\{\ldots,\ c\ \mapsto\ (q,\ cs,\ \{c_1\ \mapsto\ \square_1,\ \ldots\ ,\ c_n\ \mapsto\ \square_n\}),\ \ldots\})\ =$$
$$\Big\{\ldots\ ,\ c\ \mapsto\ \Big(\overbrace{q\ \bowtie_{iter}\ (\pi_{iter:c}(\cdot))}^{q_1},\ cs, \overset{q_1}{\lhd}(\{c_1\ \mapsto\ \square_1,\ \ldots\ ,\ c_n\ \mapsto\ \square_n\}),\ \ldots\Big)\Big\}$$

4.4.3 Aggregate

Rule

$$\frac{\begin{array}{l} ①\ \Gamma;\ loop\ \vdash e\ \Mapsto\ (q_e,\ [c],\ \emptyset) \\ ②\ q'\ \equiv\ agg_{c:<c>/iter}(q_e) \\ ③\ q_0\ \equiv (q)\ \dot{\cup}\ (@_{c:error}(loop\ \setminus\ \pi_{iter}(q'))) \\ ④\ q\ \equiv\ @_{pos:1}(q_0) \end{array}}{\Gamma;\ loop\ \vdash e.agg()\ \ \Mapsto\ (q,\ [c],\ \emptyset)}$$

$agg \in avg, max, min, sum$

Description

The Aggregate operator rules covers all the aggregate operators.

After having compiled the source in ①, the aggregated value is calculated in ②. This operation doesn't return any values for empty iterations.
If also empty iterations has to be handled, ③ adds an *error* value for them. Those are calculated as the *difference* between the loop relation and the actual result *q'*, to obtain a relation containing the iterations that doesn't have a result. After the value for empty iterations is attached, a *disjunion* is performed to merge the calculated result and the empty values.
The error value in the case of the *sum* operator it's 0 because that operator applied to an empty list returns 0, in the case of other operations an exception is generated by the actual LINQ-to-SQL provider, and an error value has to be specified.
This feature can be disabled without any problems for the compilation. To maintain absolute position, in ④ a new *pos* column is calculated. The new position value will be for every iteration 1 because the operator aggregates all the value inside each iteration in a single result.

4.4.4 Binary

Rule

$$\frac{\begin{array}{l} ① \ \Gamma;\ loop\ \vdash e_i\ \Mapsto\ (q_i,\ [c_i],\ \emptyset)|_{i=1,2} \\ ② \ cs_{2-l}\ =\ Leafs(cs_2) \qquad cs_{2-r}\ =\ Shift(cs_{2-l}, Offset(cs_{2-l})) \\ ③ \ q_r\ \equiv\ (\pi_{iter_1:iter,pos_1:pos,cs_{2-r}:cs_{2-l}}(q_2)) \\ ④ \ q\ \equiv\ \pi_{iter,\ pos,\ res}(op_{res:<cs_1;cs_{2-r}>}(q_1 \bowtie_{iter=iter_1} q_r)) \end{array}}{\Gamma;\ loop\ \vdash e_1.\circledast\ e_2()\ \Mapsto\ (q,\ [res],\ \emptyset)}$$

$$\circledast \in \{+, -, >, <, \leq, \ldots\}$$

Description

The Binary operator rules covers all type of binary operators.

In ① both operators have to be compiled. In ② some helpers function are applied to obtain the information we need to rename the columns of cs_2 because a condition for the *join* is that the name of the columns must be unique. The rename takes place in the projection in ③ while in ④ the binary operator and the final projection (*iter*, *pos* and the result column *res*) are applied.

4.4.5 Box

Rule

$$\frac{\begin{array}{l} ① \ \Gamma;\ loop\ \vdash e\ \Mapsto\ (q,\ cs,\ ts) \\ ② \ q\ \equiv\ @_{pos:1}(\pi_{iter,\ c:iter}(loop)) \end{array}}{\Gamma;\ loop\ \vdash e.Box()\ \Mapsto\ (q,\ [c],\ \{c\ \mapsto\ (q,\ cs,\ ts)\})}$$

Description

The Box rules creates a surrogate column and table to box the given relation.

The values for the surrogate column are calculated in ② and are based on the *loop* relation. The position will be always 1 on each iteration because all the values of each will be moved in the surrogate table, as we can see in *ts* in the result where we have a map from the surrogate column *c* to the original (compiled in ①) *QCsTs* struct.

4.4.6 Concat

Rule

$$\frac{\begin{array}{l} ① \; \Gamma;\; loop \vdash e_i \Mapsto (q_i,\; cs,\; ts_{e_i})|_{i=1,2} \\ ② \; q' \equiv \#_{item':<iter,ord,pos>}((@_{ord:1}(q_1)) \;\dot{\cup}\; (@_{ord:2}(q_2))) \\ ③ \; q \equiv \pi_{iter,\; pos:pos',\; notKeys(e_1),[Keys(e_1):item']}(\varrho_{pos':<ord,pos>(q')}) \end{array}}{\Gamma;\; loop \vdash e_1.Concat(e_2) \Mapsto (q,\; cs,\; \overset{q}{\oplus}(ts_1,\; ts_2))}$$

Description

The Concat rules concatenates two relations that has the same structure.

After the dispatch of the subtrees in ①, *disjunion* is performed between them with a new attached *ord* columns, which is used in the *rownum* operator to calculate the new values for the eventual surrogate columns in ②.

In ③ a *rowrank* is used to calculate the new *pos* column and the column are projected to obtain the result in the correct form. In this stage, eventually, the surrogate columns are also substituted with the new calculated values.

In the result, the ⊕ function is applied to the results *ts* to concatenate recursively also the eventual surrogate tables. As starting query is given the new *q*.

4.4.7 Conditional

Rule

$$\frac{\begin{array}{l} ① \; \Gamma;\; loop \vdash e_1 \Mapsto (q_1,\; \{c\},\; \emptyset) \\ ② \; loop_{then} \equiv \pi_{iter}(\sigma_c(q_1)) \qquad loop_{else} \equiv \pi_{iter}(\sigma_{c'}(\neg_{c':<c>}(q_1))) \\ \text{Then:} \\ ③ \; q_{v_{i then}} \equiv \pi_{iter,pos,cs_{v_i}}(q_{v_i} \bowtie_{iter=iter'} (\pi_{iter':iter}(loop_{then}))) \\ ④ \; \Gamma_{then} \equiv [\ldots,\; v_i \mapsto (q_{v_{i then}},\; cs_{v_i},\; \overset{q_{v_{i then}}}{\lhd} (ts_{v_i}))] \\ \text{Else:} \\ ③ \; q_{v_{i else}} \equiv \pi_{iter,pos,cs_{v_i}}(q_{v_i} \bowtie_{iter=iter'} (\pi_{iter':iter}(loop_{else}))) \\ ④ \; \Gamma_{else} \equiv [\ldots,\; v_i \mapsto (q_{v_{i else}},\; cs_{v_i},\; \overset{q_{v_{i else}}}{\lhd} (ts_{v_i}))] \\ ⑤ \; \Gamma_{then};\; loop_{then} \vdash e_2 \Mapsto (q_2,\; cs,\; ts_2) \qquad \Gamma_{else};\; loop_{else} \vdash e_3 \Mapsto (q_3,\; cs,\; ts_3) \\ ⑥ \; q_1 \equiv \#_{rank:<iter,ord,pos>}((@_{ord:1}(q_2) \;\dot{\cup}\; @_{ord:2}(q_3))) \\ ⑦ \; q \equiv \pi_{iter,pos,notKeys(cs),Keys(cs):rank}(q_1) \end{array}}{\{\ldots,\; v_i \mapsto (q_{v_i},\; cs_{v_i},\; ts_{v_i})\};\; loop \vdash e_1 \;?\; e_2 \;:\; e_3 \Mapsto (q,\; cs,\; ts_2 \overset{q}{\oplus} ts_3)}$$

Description

The Conditional rules compiles a conditional statement.

In ① the condition's subtree is compiled.

Then the rule take different path to generate the environments on which the *then* and the *else* subtree will be compiled. Those paths only differs on the generation of the new *loop* relation in ②: for the *then* loop are taken only the iterations inside which the condition is true, the opposite for the *else*. Since variable bindings are possible, the map needs to be forwarded for each variable inside the Γ environment based on the new loop as done in ③. This need to be done for both cases using the relative *loop* relation. In ④ the mapping is substituted in the environment and the $\lhd$ operator is applied on the *ts*. This environment with forwarded map and the new loop relation are then used to compile the respective case, as done in ⑤. In ⑥ the results of the *then* and *else* are merged with a *disjunion* operator (their *ts* will be recursively concatenated by the $\oplus$ operator on the rule's result) and a new rank column is calculated. This new rank column will be used in ⑦ to update the surrogate key columns. A new position column doesn't have to be calculated because no filtering is performed and each elements of the source will be either *true* or *false*.

4.4.8 Constant

Rule

$$\frac{q \;\equiv\; @_{item1:c}(@_{pos:1}(loop))}{\Gamma;\; loop \;\vdash c \;\; \Rrightarrow \;(q,\; [item1],\; \emptyset)}$$

Description

A constant is compiled by simple *loop lifting* it.

Its value it's the same in each iteration then its value can be attached to the *loop* relation with a *pos* column with the constant value 1 because is only one value per iteration.

4.4.9 Count

Rule

$$\frac{\begin{array}{l}\Gamma;\ loop\ \vdash e\ \Rightarrow\ (q_e,\ cs,\ ts)\\ ① \ q'\ \equiv\ count_{c:<>/iter}(q_e)\\ q_0\ \equiv (q)\ \dot{\cup}\ (@_{c:0}(loop\ \setminus\ \pi_{iter}(q')))\\ q\ \equiv\ @_{pos:1}(q_0)\end{array}}{\Gamma;\ loop\ \vdash e.Count()\ \Rightarrow\ (q,\ [c],\ \emptyset)}$$

Description

The Count operators counts the tuples of each iteration.

The rule for the Count operator is similar to the rule for the Aggregate operators shown in 4.4.3.
It differs only on the applied operator in ① because it's not calculated on the actual values inside the column but only on the number of tuples in each iteration.

4.4.10 Distinct

Rule

$$\frac{\begin{array}{l}① \ \Gamma;\ loop\ \vdash e\ \Rightarrow\ (q_e,\ cs,\ \emptyset)\\ ② \ q'\ \equiv\ \delta(\pi_{iter,\ Leafs(cs)}(q_e))\\ ③ \ q\ \equiv\ \#_{pos:<Leafs(cs)>\setminus iter}(q')\end{array}}{\Gamma;\ loop\ \vdash e.Distinct()\ \Rightarrow\ (q,\ cs,\ \emptyset)}$$

Description

The Distinct operator eliminates duplicates inside the relation.

In ① the relation is compiled and then in ② the *distinct* is applied on the compiled relation without the *pos* column. In ③ the *rownum* operator calculates the new absolute positions.

4.4.11 ElementAt

Rule

$$\frac{\begin{array}{l} ① \; \Gamma;\ loop \vdash e_1 \Mapsto (q_{e_1},\ cs,\ ts) \qquad \Gamma;\ loop \vdash e_2 \Mapsto (q_{e_2},\ \{c\},\ \emptyset) \\ ② \; q \equiv @_{pos:1}(\pi_{iter,Leafs(cs)}(==_{res:<pos,c'>}(q_{e_1} \bowtie_{iter=iter'} (\pi_{iter':iter,c':c}(q_{e_2}))))) \end{array}}{\Gamma;\ loop \vdash e_1.ElementAt(e_2) \Mapsto (q,\ cs,\ \overset{q}{\lhd} ts)}$$

Description

The ElementAt operator returns the element at a specified index. The provider supports this operator due to the strict order of the *looplifting* technique.

The subtree representing the source (e_1) and the position (e_2) are compiled in ①. To obtain the element from each iteration in ②, after a column renaming, the source and position relations are joined together on the *iter* columns and then only the tuples whose position matches the value inside the column c are taken. To ensure the right structure, a projection is performed and a new position column, with the constant value 1, because each iteration will only have 1 element, is attached.

4.4.12 Filter

Rule

$$\frac{\begin{array}{l} ① \; \Gamma;\ loop \vdash e_1 \Mapsto (q_{e_1},\ cs_{e_1},\ ts_{e_1}) \qquad \Gamma;\ loop \vdash e_2 \Mapsto (q_{e_2},\ \{c\},\ \emptyset) \\ ② \; q \equiv \#_{pos:<pos>/iter}(\pi_{iter,pos,cond}(\sigma_{c'}(q_{e_1} \bowtie_{<iter=iter',pos=pos'>} (\pi_{iter':iter,pos':pos,cond:c}(q_{e_2}))))) \end{array}}{\Gamma;\ loop \vdash e_1.FILTER(e_2) \Mapsto (q,\ cs_{e_1},\ ts_{e_1})}$$

Description

The FILTER operator filters a source based on a list of aligned boolean values.

In ① the source's (e_1) and the boolean list's (e_2) subtrees are compiled. In ②, after a renaming of the columns in q_{e_2}, a *join* is performed to bind each tuple to its corresponding boolean value. The *select* operator filters the result based on the condition column. A *project* operator recreates the correct structure and then the new absolute position column is calculated.

4.4.13 Flatten

Rule

$$\frac{\begin{array}{l} ① \; \Gamma;\ loop \vdash e \Rrightarrow (q, \{c\}, \{c \mapsto (q_c, cs_c\, ts_c)\}) \\ ② \; q_0 \equiv \varrho_{pos''':<pos'',pos>}(\pi_{iter'':iter,iter:c,pos':pos}(q) \bowtie_{iter=iter'} \pi_{iter':iter,pos,cs_c}(q_c)) \\ ③ \; q \equiv \pi_{iter:iter'',pos:pos''',cs_c}(q_0) \end{array}}{\Gamma;\ loop \vdash e.FLATTEN() \Rrightarrow (q, cs_c, ts_c)}$$

Description

The FLATTEN operator returns a list obtained by concatenating the inner lists of a relation.

In ① the subtree representing the relation that needs to be flatten is compiled. In ② the *join* on the surrogate key is performed and the new absolute position is calculated. In ③ the final projection is made.

4.4.14 GROUP

Rule

$$\frac{\begin{array}{l} ① \; \Gamma;\ loop \vdash e_k \Rrightarrow (q_k, cs_k, ts_k) \qquad \Gamma;\ loop \vdash e_v \Rrightarrow (q_v, cs_v, ts_v) \\ key \equiv Leafs(cs_k) \qquad surr \equiv Offset(cs_k) + 1 \\ ② \; q_g \equiv \pi_{iter,pos,key,surr}(\varrho_{surr:<iter,key>(q_k)}) \\ ③ \; q_{outer} \equiv \#_{pos:<key>/iter}(\delta(\pi_{iter,key,surr}(q_g))) \\ q_r \equiv \pi_{iter'=iter,pos':pos,surr}(q_g) \\ ④ \; q_{inner} \equiv \pi_{iter:surr,pos:pos'',Leafs(cs_v)}(\#_{pos'':<pos>/surr}(q_v \bowtie_{iter=iter',pos=pos'} (q_r))) \end{array}}{\Gamma;\ loop \vdash e_v.GROUP(e_k) \Rrightarrow (q_{outer}, \{Key \mapsto cs_k, surr\}, surr \mapsto (q_inner, cs_v, ts_v))}$$

Description

The GROUP operator groups two positional aligned lists.

In ① the value's and key's subtrees are compiled.

The GROUP operator returns a result in the form of nested list, with a list of keys each bounded to a list containing its grouped values. As explained in 3.5, to handle that on flat structures two queries has to be generated: a query representing the outer list and one representing the inner lists linked by surrogate values. In ②, q_g, which contains the surrogate values based on the grouping key and the *iter* column, is calculated. This relation will be used to

calculate the *inner* and *outer* queries. The outer query, q_{outer}, is q_g with no duplicates and a new position column as shown in ③.

q_{inner} represents the surrogate table and is where the grouping takes place with a *thetajoin* between q_g, which contains the grouping key, and the relation representing the values in ④. The new positional column is calculated and in the final projection the mapping with the *outer* query is done, by using the surrogate key column of q_g as *iter*.

4.4.15 Member Access

Rule

$$\frac{\begin{array}{l} ①\ \Gamma;\ loop\ \vdash e\ \Rightarrow\ (q_e,\ cs\ \equiv\ \{\ldots,\ f\ \mapsto\ cs_f,\ \ldots\},\ ts) \\ ②\ c\ \equiv\ cardBefore(cs,\ f) \\ c_{ids}\ \equiv\ Leafs(cs_f) \qquad cs'_f\ \equiv\ decr(cs_f,\ c) \qquad c'_{ids}\ \equiv\ Leafs(cs'_f) \\ ③\ ts'\ \equiv\ decr(retrByKeys(ts, c_{ids}),\ c) \\ ④\ q\ \equiv\ \pi_{iter,pos,c'_{ids}:c_{ids}}(q_e) \end{array}}{\Gamma;\ loop\ \vdash e.f\ \Rightarrow\ (q,\ cs'_f,\ ts')}$$

Description

The Member Access represents the access to a member of a relation.

The subtree is compiled in ① and in the result's *cs* there must be a map to the accessed member. Due to the fact that a member access changes the scope of what can be seen after it, some shifting operations on the column names are needed because after it only what's inside the member can be accessed. In ② some variables are calculated: the cardinality *c* represents to which column the member is linked and it's used to shift the cs_f structure, which it's the internal structure of the member, backwards, in order to have its columns starting from 1. The cs'_f represents the structure of what can be accessed after. In ③ the *ts* structure is shifted backward according to the cs_f's shifting, so the eventual surrogate columns match as before. In ④ the final structure is built, with the old columns been replaced by the new shifted ones.

4.4.16 Not

Rule

$$\frac{\begin{array}{l} ① \; \Gamma;\ loop\ \vdash e\ \mapsto\ (q,\ \{c\},\ \emptyset) \\ ② \; q\ \equiv\ \pi_{iter,pos,c:res}(\neg_{res:<c>}(q)) \end{array}}{\Gamma;\ loop\ \vdash !e\ \mapsto\ (q,\ \{c\},\ \emptyset)}$$

Description

The Not operator returns the boolean negation of the given relation. The relation must contain only one boolean column which is not a surrogate column.

After the subtree representing the relation is compiled in ①, the *not* operator is applied in ②. The projection recreates to the correct structure. No new position column has to be calculated because the number and order of tuples in each iteration are not modified.

4.4.17 Record

Rule

$$\frac{\begin{array}{l} \Gamma;\ loop\ \vdash\ new\{f_1\ =\ e_1\} \mapsto\ (q_1,\ cs_1,\ ts_1) \\ \Gamma;\ loop\ \vdash\ new\{f_1\ =\ e'_1\ \ldots\ f_n\ =\ e'_n\} \mapsto\ (q_2,\ cs_2,\ ts_2) \\ cols_1\ \equiv\ Leafs(cs1) \qquad cols_2\ \equiv\ Leafs(cs_2) \\ cols'_2\ \equiv\ Shift(cols_2,\ Offset(cols_1)) \qquad ts'_2\ \equiv\ Shift(ts_2,\ Offset(cols_1)) \\ q\ \equiv\ \pi_{iter,pos,cols_1,cols'_2}(q_1\ \bowtie_{iter=iter'}\ (\pi_{iter':iter,cols'_2:cols_2}(q_2))) \end{array}}{\Gamma;\ loop\ \vdash new\left\{\begin{array}{c} f_1\ =\ e'_1 \\ \vdots \\ f_n\ =\ e'_n \end{array}\right\}\ \mapsto\ (q,\ cs_1 \parallel cs_2,\ ts'_2)}$$

$$\frac{\Gamma;\ loop\ \vdash\ e\ \mapsto\ (q,\ cs,\ ts)}{\Gamma;\ loop\ \vdash new\left\{f\ =\ e\right\}\ \mapsto\ (q,\ \{\ f\ \mapsto\ cs\},\ ts)}$$

Description

The Record rule handles the definition of a new object by mapping its structure to a flat relational structure.

The structures generated by the independent compilation of each parameter must be combined to obtain the relational representation of the new object.

Two rules are used to emulate a loop where the structure resulting from the compilation of the parameter's subtree is concatenated with the structures generated by the the other parameters. Only shifts are performed in order to ensure the uniqueness of the indexes.

An example is shown in figure 4.3

4.4.18 Select

Rule

$$\frac{\begin{array}{l} ① \Gamma;\ loop \vdash e_1 \Mapsto (q_1,\ cs_1,\ ts_1) \\ ② s \equiv \#_{inner:<iter,pos>}(q_1) \\ ③ q_v \equiv @_{pos:1}(\pi_{iter:inner,Leafs(cs_1)}(s)) \\ ④ loop_v \equiv \pi_{iter}(q_v) \qquad map \equiv \pi_{outer:iter,inner}(s) \\ ⑤ \Gamma_v \equiv [\ldots,\ v_i \mapsto (\pi_{iter:inner,pos,Leafs(cs_{v_i})}(q_{v_i} \bowtie_{iter=outer} map),\ cs_{v_i},\ ts_{v_i}), \ldots] \\ ⑥ \Gamma_v + (v \mapsto (q_v,\ cs_1,\ ts_1)),\ loop_v \vdash e_2 \Mapsto (q_2,\ cs_2,\ ts_2) \\ ⑤ q \equiv \pi_{iter:outer,pos:pos',Leafs(cs_2)}(\#_{pos':<iter,pos>/outer}(q_2 \bowtie_{iter=inner} map)) \end{array}}{\Gamma : [\ldots,\ v_i \mapsto (q_i,\ cs_i,\ ts_i), \ldots];\ loop \vdash e1.Select(v \Rightarrow e_2) \Mapsto (q,\ cs_2,\ ts_2)}$$

Description

The Select rule handles a Select operation over a given source.

In ① the source is compiled. Out of it, in ②, new surrogate value are calculated for later use. In ③ the *loop-lifted* representation of the variable v, which at each iteration will contain an element of the source, is calculated. Its positional values are replaced by a 1, because every item is now a singleton sequence within its iteration. In ④ a new *loop* relation and the *map* used for back mapping between scopes are calculated. Using this new infos, in ⑤, a forward mapping is performed on each variable inside the Γ environment since its representation depends on the *loop* iteration and that's changed due to the Select operator. In ⑥ the lambda expression is compiled in the re-mapped Γ environment, where also the *loop lifted v* variable is added. In ⑦ the resulting plan is back-mapped into the parent scope.

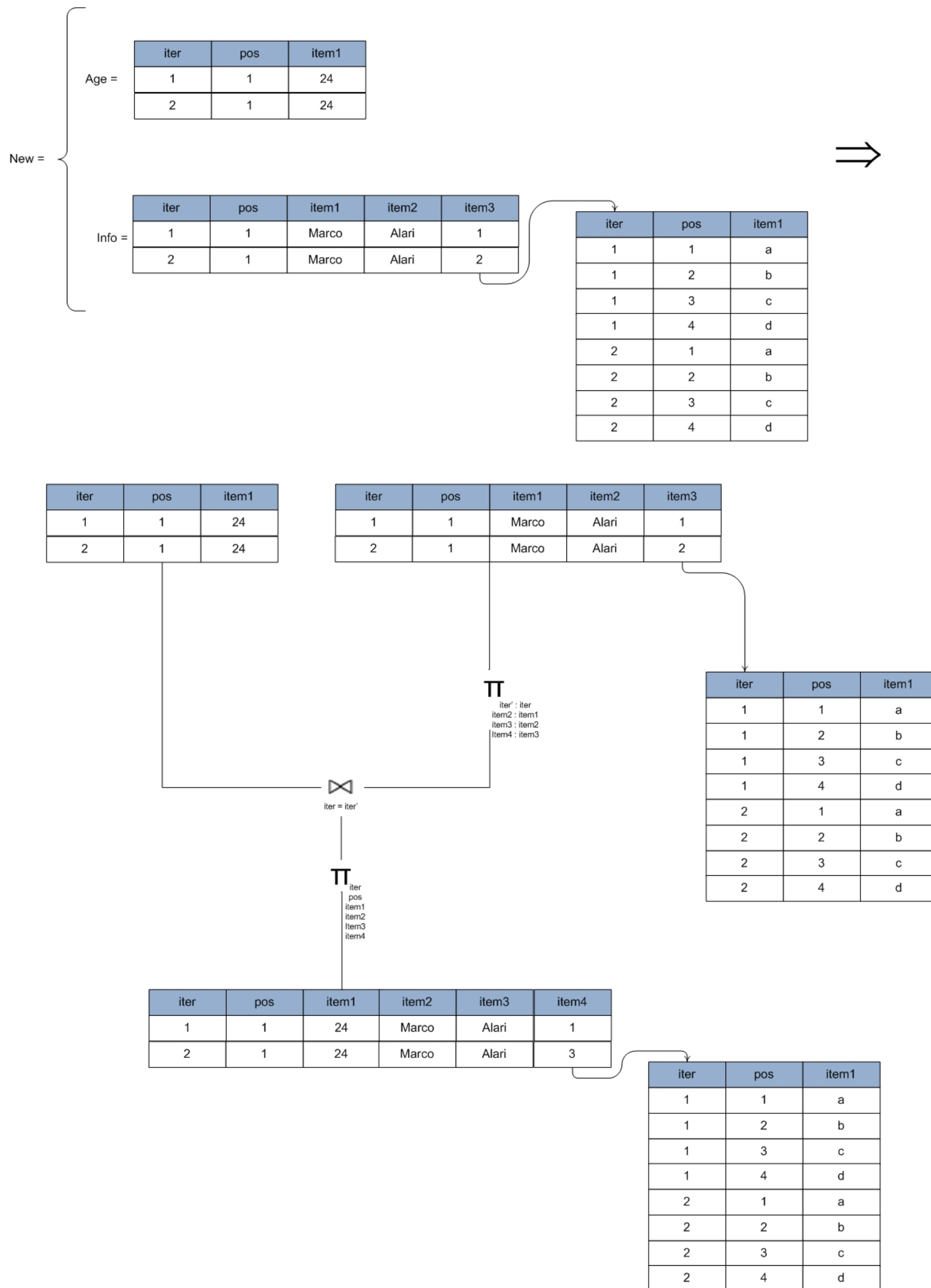


Figure 4.3: Example of the record compilation rule

4.4.19 Single

Rule

$$\frac{\Gamma;\ loop\ \vdash e\ \Mapsto\ (q,\ cs,\ ts)}{\Gamma;\ loop\ \vdash e.Single()\ \Mapsto\ (q,\ cs,\ ts)}$$

Description

The Single returns the only element of a list.

It is just a dispatch rule because relationally, the representation of a singleton list containing e and the the pure e are alike. The subtree is compiled and its result is the rule's result.

4.4.20 Skip

Rule

$$\frac{\begin{array}{c}\Gamma;\ loop\ \vdash e_1\ \Mapsto\ (q_{e_1},\ cs,\ ts) \qquad \Gamma;\ loop\ \vdash e_2\ \Mapsto\ (q_{e_2},\ \{c\},\ \emptyset) \\ \textcircled{1}\ q\ \equiv\ \pi_{iter,pos':pos,Leafs(cs)}(\varrho_{pos':<pos>}(\sigma_{res}(>_{res:<pos,c'>}\ (q_{e_1}\ \bowtie_{iter=iter'}\ (\pi_{iter':iter,c':c}(q_{e_2}))))))\end{array}}{\Gamma;\ loop\ \vdash e_1.Skip(e_2)\ \Mapsto\ (q,\ cs,\ \overset{q}{\lhd} ts)}$$

Description

The Skip operator uses positional access to return the input source without the first n elements.

It's similar to the ElementAt rule showed in section 4.4.11, but in that case all the tuples of each iteration with a position greater than the threshold are taken, therefore as comparison between position and threshold a > is used instead of a ==. Since it returns a list of elements, also new positions must be calculated. The $\lhd$ operator is used to polish the result's ts.

4.4.21 Table

Rule

$$\frac{\textcircled{1}\ q\ \equiv\ loop\ \times\ \varrho_{pos:<c_{k_1},\ldots,c_{k_n}>}(\text{🛢}_R)}{\Gamma;\ loop\ \vdash Table\ R\ (c_1,\ldots,c_n)\ with\ keys\ (c_{k_1},\ldots,c_{k_n})\ \Mapsto\ (q,\ \{c_1\ \mapsto\ 1,\ldots,c_n\ \mapsto\ n\},\ \emptyset)}$$

Description

The Table rule compiles a reference to a database resident table.

In ① a new positional column is calculated on the table. The *cross product* with the *loop* relation is done to *loop lift* it.

4.4.22 Take

Rule

$$\frac{\begin{array}{l} ①\ \Gamma;\ loop\ \vdash e_1\ \Rightarrow\ (q_{e_1},\ cs,\ ts) \qquad \Gamma;\ loop\ \vdash e_2\ \Rightarrow\ (q_{e_2},\ \{c\},\ \emptyset) \\ ②\ q\ \equiv\ \pi_{iter,pos':pos,Leafs(cs)}(\varrho_{pos':<pos>}(\sigma_{res}(\leq_{res:<pos,c'>}\ (q_{e_1}\ \bowtie_{iter=iter'}\ (\pi_{iter':iter,c':c}(q_{e_2})))))) \end{array}}{\Gamma;\ loop\ \vdash e_1.Take(e_2)\ \Rightarrow\ (q,\ cs,\ \overset{q}{\triangleleft} ts)}$$

Description

The Take operators returns the first n elements of a sequence.

The rule is similar to the Skip rule shown in section 4.4.11, except that since from each iteration the tuples whose position is under the threshold has to be taken. In the comparison between tuple's position and threshold a $\leq$ is used instead of a $==$.

4.4.23 UnBox

Rule

$$\frac{\begin{array}{l} ①\ \Gamma;\ loop\ \vdash e\ \Rightarrow\ (q,\ [c], \{\ c\ \mapsto\ \ (q_c,\ cs_c,\ ts_c)\}) \\ ②\ q_0\ \equiv\ \pi_{iter:iter',pos,Leafs(cs_c)}(\pi iter' : iter, iter : c(q)\ \bowtie_{iter=iter'}\ (\pi_{iter,Leafs(cs_c)}(q_c))) \end{array}}{\Gamma;\ loop\ \vdash e.UnBox()\ \Rightarrow\ (q_0,\ cs_c,\ ts_c)}$$

Description

The UnBox operator un-boxes a previously boxed subtree.

In ① the subtree is compiled. In the result there must be only a surrogate column c with a corresponding mapping inside the *ts* to its surrogate table. In ② the *un-boxing* join between the surrogate column and the *iter* column of the surrogate table is performed and the final projection is done.

4.4.24 Variable

Rule

$$\frac{}{\{\ldots, v \mapsto (q, cs, ts), \ldots\}; loop \vdash v \Mapsto (q, cs, ts)}$$

Description

The variable rule un-nests its informations from the Γ environment and returns what its *loop lifted* value.

4.5 The Running Example

4.5.1 The algebraic plan

In figure 4.4 the algebraic plan produced by the compilation is shown. Due to the enormous amount of operators the figure is un-intelligible.

In figure 4.5 is shown in details the number of generated queries ans the structure of the resulting relations.

Q_0 represents the outer query with the record structure containing the name of the customer, its region and its orders. The *order* parameter is a nested list therefore handled with *surrogate values*. The inner query representing its values is Q_1, which resembles the record containing the status (which is the grouping key), and the two lists with the related orders infos: the record containing price and date and the number of items of the order. Both are again nested lists, and their inner values are represented by Q_2 and Q_3 respectively.

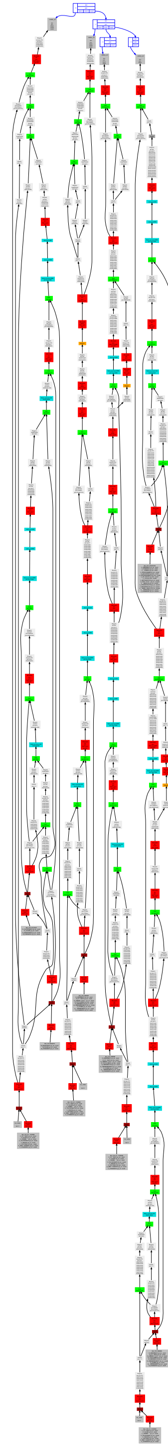


Figure 4.4: The algebraic plan of the query in 3.6.1

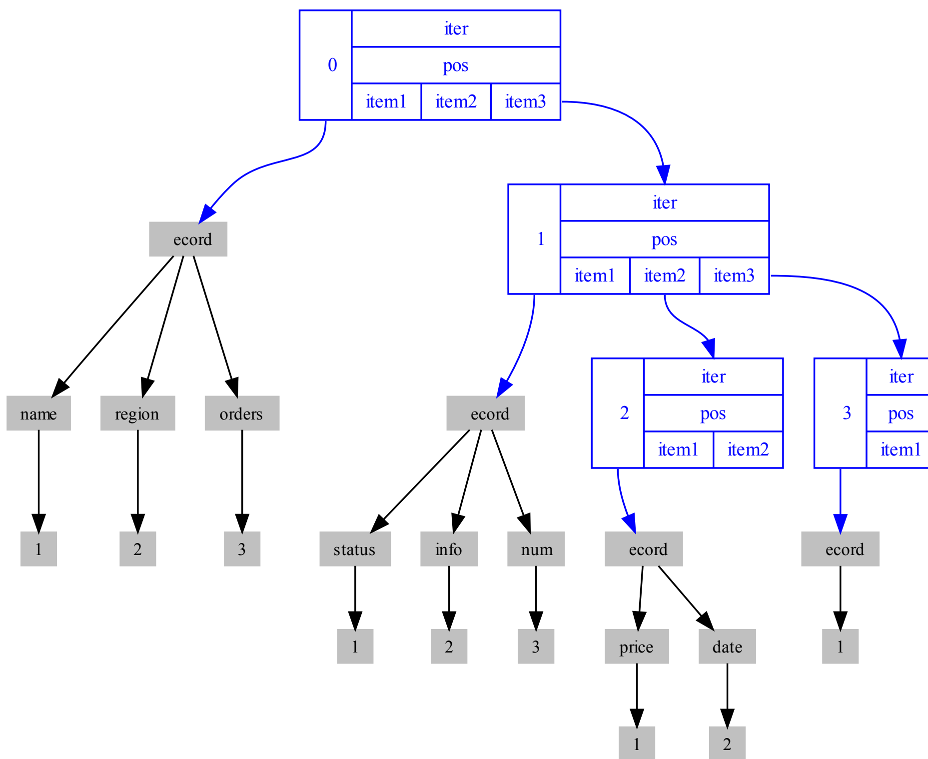


Figure 4.5: The end result structure of the query in 3.6.1

Chapter 5

We have to go back

Now that the algebraic plan is ready the only thing left to do is to generate the SQL code. In this phase the Pathfinder engine will do all the work, leaving to us only to execute the SQL code on the database and to bring the result back into the object type the user wants. This last part is however out of the focus of this thesis due to its dimensions and totally different problematics that may arise.

The xml representation of the generated algebraic plan created in section 5.1 will be given to the Pathfinder engine to be optimized. Out of this the Pathfinder SQL code generator will finally produce the SQL code that will be executed via ODBC on the DBMS. A cursor to the results inside the heap will be given to us.

5.1 Xml representation

The compiled algebraic tree must be represented as an xml tree, following the specs shown in the Pathfinder wiki, http://wiki.pathfinder-xquery.org/wiki/index.php/Algebra_XML_Output.

A query plan bundle must be created, containing a query plan for each algebraic tree, including all the query inside the *ts* structure at every level of nesting. The nesting level and the dependencies between each of them must be maintained and the inner queries must be linked to theirs outer query.

5.2 PathFinder

The Pathfinder engine [1] will be used for both optimizing [7] and then generate the SQL code [2].

The whole query plan bundle must be given to PathFinder and an optimized version of it will be produced.

This optimized version must be again given to PathFinder in order to generate the SQL code. The SQL code will contain a SQL statement for each query plan contained in the given query plan bundle. This allows us to only process the query for a certain nesting level only when it is accessed.

5.3 Heap results

Through ODBC the generated SQL code can be executed against the wished database. A connection for each execution must be established and on it the SQL statement executed.

The execution will give back a pointer to the result in the heap. The results must be then back-mapped into the structure defined by the user and eventual changes on the data must be reproduced also on the database. This operations are however out of the scope of this thesis, due to their huge dimensions and the totally different type of problems that may arise. Meta programming is needed to define an object type at run type and then a huge problem to solve is to instantiate maybe millions of objects at once, or find a way to overcome that.

5.4 The Running Example

5.4.1 The optimized plan

In figure 5.1 the algebraic plan optimized by the Pathfinder engine is shown. It's possible to see how compact is compared to the not optimized version in figure 4.4

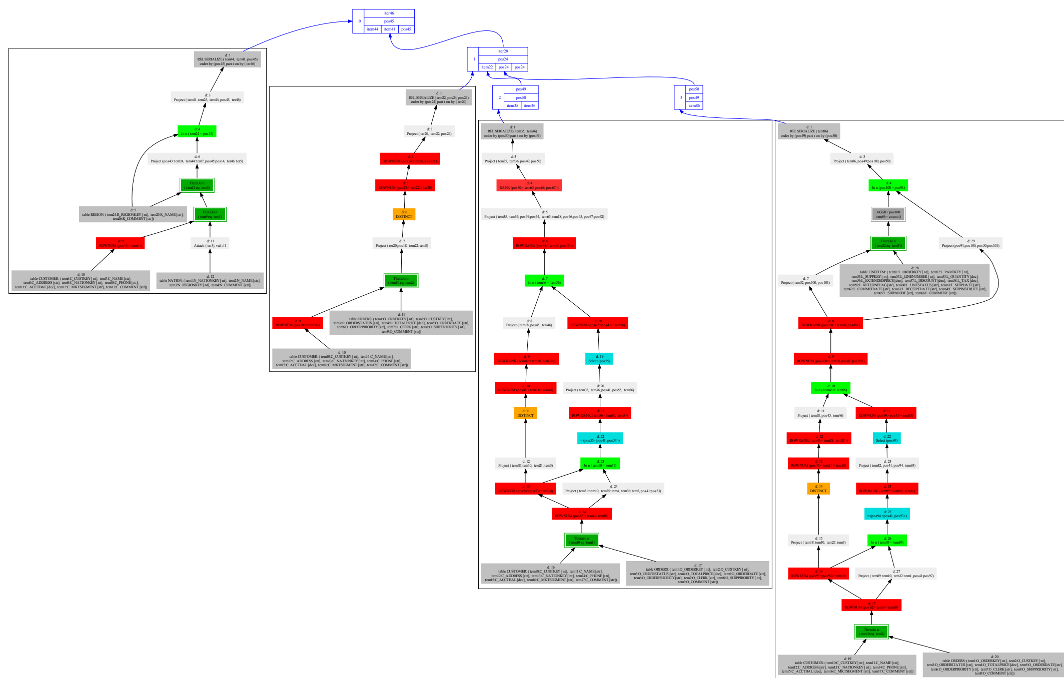


Figure 5.1: The optimized version of the compiled tree showed in figure 4.4

5.4.2 The generated SQL code

Q_0

Q_0 represents the outer query with the record structure containing the name of the customer, its region and its orders. The *order* parameter is a nested list therefore is a surrogate column.

```
1 WITH
2 -- binding due to rownum operator
3 t0000 (item6_int, item7_str, item8_str, item9_int, item10_str,
4   item11_dec, item12_str, item13_str, pos14_nat) AS
5   (SELECT a0001.C_CUSTKEY AS item6_int, a0001.C_NAME AS item7_str,
6           a0001.C_ADDRESS AS item8_str, a0001.C_NATIONKEY AS item9_int,
7           a0001.C_PHONE AS item10_str, a0001.C_ACCTBAL AS item11_dec,
```

```
 8            a0001.C_MKTSEGMENT AS item12_str, a0001.C_COMMENT AS item13_str,
 9            ROW_NUMBER () OVER (ORDER BY a0001.C_CUSTKEY ASC) AS pos14_nat
10       FROM CUSTOMER AS a0001)
11
12 SELECT 1 AS iter29_nat,
13          a0002.pos14_nat AS pos27_nat,
14          a0002.item7_str AS item26_str, a0000.R_NAME AS item25_str
15     FROM REGION AS a0000,
16          t0000 AS a0002,
17          NATION AS a0003
18   WHERE a0002.item9_int = a0003.N_NATIONKEY
19     AND a0000.R_REGIONKEY = a0003.N_REGIONKEY
20   ORDER BY a0002.pos14_nat ASC;
```

Q_1

The query Q_1 represents the inner values of the outer *orders* column of Q_0. It resembles the record containing the status (which is the grouping key), and the two lists with the related orders infos which are again surrogate columns.

```
 1 WITH
 2 -- binding due to rownum operator
 3 t0000 (item10_int, item11_str, item12_str, item13_int, item14_str,
 4   item15_dec, item16_str, item17_str, pos18_nat) AS
 5   (SELECT a0000.C_CUSTKEY AS item10_int, a0000.C_NAME AS item11_str,
 6            a0000.C_ADDRESS AS item12_str, a0000.C_NATIONKEY AS item13_int,
 7            a0000.C_PHONE AS item14_str, a0000.C_ACCTBAL AS item15_dec,
 8            a0000.C_MKTSEGMENT AS item16_str, a0000.C_COMMENT AS item17_str,
 9            ROW_NUMBER () OVER (ORDER BY a0000.C_CUSTKEY ASC) AS pos18_nat
10       FROM CUSTOMER AS a0000),
11
12 -- binding due to duplicate elimination
13 t0001 (pos20_nat, item22_str) AS
14   (SELECT DISTINCT a0001.pos18_nat AS pos20_nat, a0002.O_ORDERSTATUS AS
       item22_str
15       FROM t0000 AS a0001,
16            ORDERS AS a0002
17      WHERE a0001.item10_int = a0002.O_CUSTKEY),
18
19 -- binding due to rownum operator
20 t0002 (pos20_nat, item22_str, pos23_nat) AS
21   (SELECT a0003.pos20_nat, a0003.item22_str,
22            ROW_NUMBER () OVER
```

```
23            (PARTITION BY a0003.pos20_nat ORDER BY a0003.item22_str ASC) AS
24            pos23_nat
25       FROM t0001 AS a0003)
26
27 SELECT ROW_NUMBER () OVER (ORDER BY a0004.pos20_nat ASC, a0004.pos23_nat ASC
     )
28          AS pos24_nat, a0004.item22_str, a0004.pos20_nat
29     FROM t0002 AS a0004
30   ORDER BY a0004.pos20_nat ASC, a0004.pos20_nat ASC, a0004.pos23_nat ASC;
```

Q_2

The query Q_2 represents the inner values of the outer *info* column of query Q_1. It contains the price and order date of each order.

```
1  WITH
2  -- binding due to rownum operator
3  t0000 (item10_int, item11_str, item12_str, item13_int, item14_str,
4    item15_dec, item16_str, item17_str, item1_int, item2_int, item3_str,
5    item4_dec, item5_str, item6_str, item7_str, item8_int, item9_str,
6    pos106_nat) AS
7    (SELECT a0000.C_CUSTKEY AS item10_int, a0000.C_NAME AS item11_str,
8            a0000.C_ADDRESS AS item12_str, a0000.C_NATIONKEY AS item13_int,
9            a0000.C_PHONE AS item14_str, a0000.C_ACCTBAL AS item15_dec,
10           a0000.C_MKTSEGMENT AS item16_str, a0000.C_COMMENT AS item17_str,
11           a0001.O_ORDERKEY AS item1_int, a0001.O_CUSTKEY AS item2_int,
12           a0001.O_ORDERSTATUS AS item3_str, a0001.O_TOTALPRICE AS item4_dec,
13           a0001.O_ORDERDATE AS item5_str, a0001.O_ORDERPRIORITY AS item6_str
               ,
14           a0001.O_CLERK AS item7_str, a0001.O_SHIPPRIORITY AS item8_int,
15           a0001.O_COMMENT AS item9_str,
16           ROW_NUMBER () OVER
17           (PARTITION BY a0000.C_CUSTKEY ORDER BY a0001.O_ORDERKEY ASC) AS
18           pos106_nat
19      FROM CUSTOMER AS a0000,
20           ORDERS AS a0001
21     WHERE a0000.C_CUSTKEY = a0001.O_CUSTKEY),
22
23 -- binding due to rownum operator
24 t0001 (item10_int, item11_str, item12_str, item13_int, item14_str,
25   item15_dec, item16_str, item17_str, item1_int, item2_int, item3_str,
26   item4_dec, item5_str, item6_str, item7_str, item8_int, item9_str,
27   pos106_nat, pos107_nat) AS
28   (SELECT a0002.item10_int, a0002.item11_str, a0002.item12_str,
```

```
29            a0002.item13_int, a0002.item14_str, a0002.item15_dec,
30            a0002.item16_str, a0002.item17_str, a0002.item1_int, a0002.
                  item2_int,
31            a0002.item3_str, a0002.item4_dec, a0002.item5_str, a0002.item6_str
                  ,
32            a0002.item7_str, a0002.item8_int, a0002.item9_str, a0002.
                  pos106_nat,
33            ROW_NUMBER () OVER
34            (PARTITION BY a0002.item10_int ORDER BY a0002.pos106_nat ASC) AS
35            pos107_nat
36      FROM t0000 AS a0002),
37
38 -- binding due to duplicate elimination
39 t0002 (item18_int, item28_str) AS
40   (SELECT DISTINCT a0003.item10_int AS item18_int, a0003.item3_str AS
         item28_str
41      FROM t0001 AS a0003),
42
43 -- binding due to rownum operator
44 t0003 (item18_int, item28_str, pos59_nat) AS
45   (SELECT a0004.item18_int, a0004.item28_str,
46            ROW_NUMBER () OVER
47            (PARTITION BY a0004.item18_int ORDER BY a0004.item28_str ASC) AS
48            pos59_nat
49      FROM t0002 AS a0004),
50
51 -- binding due to rank operator
52 t0004 (item18_int, item28_str, pos59_nat, item60_nat) AS
53   (SELECT a0005.item18_int, a0005.item28_str, a0005.pos59_nat,
54            DENSE_RANK () OVER
55            (ORDER BY a0005.item18_int ASC, a0005.item28_str ASC) AS
                  item60_nat
56      FROM t0003 AS a0005),
57
58 -- binding due to rank operator
59 t0005 (item10_int, item11_str, item12_str, item13_int, item14_str,
60   item15_dec, item16_str, item17_str, item1_int, item2_int, item3_str,
61   item4_dec, item5_str, item6_str, item7_str, item8_int, item9_str,
62   pos106_nat, pos107_nat, item103_int, item46_int, pos55_nat,
63   pos108_bool, item109_nat) AS
64   (SELECT a0007.item10_int, a0007.item11_str, a0007.item12_str,
65            a0007.item13_int, a0007.item14_str, a0007.item15_dec,
66            a0007.item16_str, a0007.item17_str, a0007.item1_int, a0007.
                  item2_int,
67            a0007.item3_str, a0007.item4_dec, a0007.item5_str, a0007.item6_str
```

```
                ,
68          a0007.item7_str, a0007.item8_int, a0007.item9_str, a0007.
                pos106_nat,
69          a0007.pos107_nat, a0008.item10_int AS item103_int,
70          a0008.item1_int AS item46_int, a0008.pos106_nat AS pos55_nat,
71          CASE WHEN a0008.pos106_nat = a0007.pos107_nat THEN 1 ELSE 0 END AS
72          pos108_bool,
73          DENSE_RANK () OVER
74          (ORDER BY a0007.item10_int ASC, a0007.item3_str ASC) AS
                item109_nat
75      FROM t0001 AS a0007,
76          t0000 AS a0008
77     WHERE a0007.item10_int = a0008.item10_int),
78
79  -- binding due to rownum operator
80  t0006 (item46_int, pos55_nat, pos108_bool, item109_nat, pos113_nat) AS
81    (SELECT a0009.item46_int, a0009.pos55_nat, a0009.pos108_bool,
82          a0009.item109_nat,
83          ROW_NUMBER () OVER
84          (PARTITION BY a0009.item109_nat ORDER BY a0009.pos55_nat ASC) AS
85          pos113_nat
86      FROM t0005 AS a0009
87     WHERE a0009.pos108_bool = 1),
88
89  -- binding due to rownum operator
90  t0007 (item18_int, pos59_nat, item60_nat, item46_int, pos55_nat,
91    pos108_bool, item109_nat, pos113_nat, pos114_nat) AS
92    (SELECT a0006.item18_int, a0006.pos59_nat, a0006.item60_nat, a0010.
         item46_int,
93          a0010.pos55_nat, a0010.pos108_bool, a0010.item109_nat,
94          a0010.pos113_nat,
95          ROW_NUMBER () OVER
96          (ORDER BY a0006.item18_int ASC, a0006.pos59_nat ASC, a0010.
                pos113_nat
97           ASC) AS pos114_nat
98      FROM t0004 AS a0006,
99          t0006 AS a0010
100    WHERE a0006.item60_nat = a0010.item109_nat),
101
102 -- binding due to rank operator
103 t0008 (item18_int, pos59_nat, item60_nat, item46_int, pos55_nat,
104   pos108_bool, item109_nat, pos113_nat, pos114_nat, pos115_nat) AS
105   (SELECT a0011.item18_int, a0011.pos59_nat, a0011.item60_nat, a0011.
         item46_int,
106         a0011.pos55_nat, a0011.pos108_bool, a0011.item109_nat,
```

```
107             a0011.pos113_nat, a0011.pos114_nat,
108             DENSE_RANK () OVER
109             (ORDER BY a0011.item18_int ASC, a0011.pos59_nat ASC) AS pos115_nat
110       FROM t0007 AS a0011),
111
112 -- binding due to aggregate
113 t0009 (pos114_nat, item100_int) AS
114    (SELECT a0012.pos114_nat, COUNT (*) AS item100_int
115       FROM t0008 AS a0012,
116             LINEITEM AS a0013
117      WHERE a0012.item46_int = a0013.L_ORDERKEY
118      GROUP BY a0012.pos114_nat)
119
120 SELECT a0015.pos115_nat AS pos64_nat, a0014.item100_int
121     FROM t0009 AS a0014,
122          t0008 AS a0015
123    WHERE a0014.pos114_nat = a0015.pos114_nat
124    ORDER BY a0015.pos115_nat ASC, a0014.pos114_nat ASC;
```

Q_3

The query Q_3 represents the inner values of the outer *num* column of query Q_1. It contains the number of items of each order.

```
1  WITH
2  -- binding due to rownum operator
3  t0000 (item10_int, item11_str, item12_str, item13_int, item14_str,
4    item15_dec, item16_str, item17_str, item1_int, item2_int, item3_str,
5    item4_dec, item5_str, item6_str, item7_str, item8_int, item9_str,
6    pos67_nat) AS
7    (SELECT a0000.C_CUSTKEY AS item10_int, a0000.C_NAME AS item11_str,
8             a0000.C_ADDRESS AS item12_str, a0000.C_NATIONKEY AS item13_int,
9             a0000.C_PHONE AS item14_str, a0000.C_ACCTBAL AS item15_dec,
10            a0000.C_MKTSEGMENT AS item16_str, a0000.C_COMMENT AS item17_str,
11            a0001.O_ORDERKEY AS item1_int, a0001.O_CUSTKEY AS item2_int,
12            a0001.O_ORDERSTATUS AS item3_str, a0001.O_TOTALPRICE AS item4_dec,
13            a0001.O_ORDERDATE AS item5_str, a0001.O_ORDERPRIORITY AS item6_str
                ,
14            a0001.O_CLERK AS item7_str, a0001.O_SHIPPRIORITY AS item8_int,
15            a0001.O_COMMENT AS item9_str,
16            ROW_NUMBER () OVER
17            (PARTITION BY a0000.C_CUSTKEY ORDER BY a0001.O_ORDERKEY ASC) AS
18            pos67_nat
19      FROM CUSTOMER AS a0000,
```

```
20            ORDERS AS a0001
21      WHERE a0000.C_CUSTKEY = a0001.O_CUSTKEY),
22
23 -- binding due to rownum operator
24 t0001 (item10_int, item11_str, item12_str, item13_int, item14_str,
25   item15_dec, item16_str, item17_str, item1_int, item2_int, item3_str,
26   item4_dec, item5_str, item6_str, item7_str, item8_int, item9_str,
27   pos67_nat, pos68_nat) AS
28   (SELECT a0002.item10_int, a0002.item11_str, a0002.item12_str,
29           a0002.item13_int, a0002.item14_str, a0002.item15_dec,
30           a0002.item16_str, a0002.item17_str, a0002.item1_int, a0002.
               item2_int,
31           a0002.item3_str, a0002.item4_dec, a0002.item5_str, a0002.item6_str
               ,
32           a0002.item7_str, a0002.item8_int, a0002.item9_str, a0002.pos67_nat
               ,
33           ROW_NUMBER () OVER
34           (PARTITION BY a0002.item10_int ORDER BY a0002.pos67_nat ASC) AS
35           pos68_nat
36      FROM t0000 AS a0002),
37
38 -- binding due to duplicate elimination
39 t0002 (item18_int, item28_str) AS
40   (SELECT DISTINCT a0003.item10_int AS item18_int, a0003.item3_str AS
        item28_str
41      FROM t0001 AS a0003),
42
43 -- binding due to rownum operator
44 t0003 (item18_int, item28_str, pos59_nat) AS
45   (SELECT a0004.item18_int, a0004.item28_str,
46           ROW_NUMBER () OVER
47           (PARTITION BY a0004.item18_int ORDER BY a0004.item28_str ASC) AS
48           pos59_nat
49      FROM t0002 AS a0004),
50
51 -- binding due to rank operator
52 t0004 (item18_int, item28_str, pos59_nat, item60_nat) AS
53   (SELECT a0005.item18_int, a0005.item28_str, a0005.pos59_nat,
54           DENSE_RANK () OVER
55           (ORDER BY a0005.item18_int ASC, a0005.item28_str ASC) AS
               item60_nat
56      FROM t0003 AS a0005),
57
58 -- binding due to rank operator
59 t0005 (item10_int, item11_str, item12_str, item13_int, item14_str,
```

```
60   item15_dec, item16_str, item17_str, item1_int, item2_int, item3_str,
61   item4_dec, item5_str, item6_str, item7_str, item8_int, item9_str,
62   pos67_nat, pos68_nat, item65_int, item49_dec, item50_str, pos55_nat,
63   pos69_bool, item70_nat) AS
64   (SELECT a0007.item10_int, a0007.item11_str, a0007.item12_str,
65           a0007.item13_int, a0007.item14_str, a0007.item15_dec,
66           a0007.item16_str, a0007.item17_str, a0007.item1_int, a0007.
                item2_int,
67           a0007.item3_str, a0007.item4_dec, a0007.item5_str, a0007.item6_str
                ,
68           a0007.item7_str, a0007.item8_int, a0007.item9_str, a0007.pos67_nat
                ,
69           a0007.pos68_nat, a0008.item10_int AS item65_int,
70           a0008.item4_dec AS item49_dec, a0008.item5_str AS item50_str,
71           a0008.pos67_nat AS pos55_nat,
72           CASE WHEN a0008.pos67_nat = a0007.pos68_nat THEN 1 ELSE 0 END AS
73           pos69_bool,
74           DENSE_RANK () OVER
75           (ORDER BY a0007.item10_int ASC, a0007.item3_str ASC) AS item70_nat
76      FROM t0001 AS a0007,
77           t0000 AS a0008
78     WHERE a0007.item10_int = a0008.item10_int),
79
80 -- binding due to rownum operator
81 t0006 (item49_dec, item50_str, pos55_nat, pos69_bool, item70_nat,
82   pos76_nat) AS
83   (SELECT a0009.item49_dec, a0009.item50_str, a0009.pos55_nat, a0009.
       pos69_bool,
84           a0009.item70_nat,
85           ROW_NUMBER () OVER
86           (PARTITION BY a0009.item70_nat ORDER BY a0009.pos55_nat ASC) AS
87           pos76_nat
88      FROM t0005 AS a0009
89     WHERE a0009.pos69_bool = 1)
90
91 SELECT DENSE_RANK () OVER
92        (ORDER BY a0006.item18_int ASC, a0006.pos59_nat ASC) AS pos63_nat,
93        a0010.item50_str, a0010.item49_dec
94   FROM t0004 AS a0006,
95        t0006 AS a0010
96  WHERE a0006.item60_nat = a0010.item70_nat
97  ORDER BY a0006.item18_int ASC, a0006.pos59_nat ASC, a0006.item18_int ASC,
98        a0006.pos59_nat ASC, a0010.pos76_nat ASC;
```

Chapter 6

Replacing the Leader

In this chapter will be shown how to build a provider and how to use it to execute a query instead of the original Microsoft one.

In 6.1 is shown which interfaces the provider class has to implement and how to implement a basic provider. The last step, shown in 6.2, is to generate a data context that represents in the object world the queried database and that forces the C# compiler to use our provider.

6.1 Creating the provider

6.1.1 Overview of the interfaces

A provider is a class that implements the methods of the *IQueryable* interface. In fact, the *IQueryable* interface is splitted into 2 different interfaces, the weakly typed IQueryable (and its typed version IQueryable<T>) and the IQueryProvider interface. So what need to be implemented are both the *IQueryable* interface and the IQueryProvider.

The IQueryable interface contains only read-only proprieties that can be interesting for the user. The non-generic IQueryable interface was introduced to give a weakly typed entry point for dynamic query building where the returning type is not statically known. The typed IQueryable<T> has to be implemented for a often used object type <T>.
IQueryProvider represents the actual implementation of the provider referenced by the IQueryable instance. It implements the methods to parse the expression tree and evaluate it.

A new provider has to extend the IQueryable<T> interface, because, has we see, the weakly typed IQueryable interface is already implemented inside that.

The definition of the IQueryable<T> interface is:

```
1 public interface IQueryable<T> : IEnumerable<T>,
2         IQueryable, IEnumerable
3     {
4     }
```

It doesn't have any specific method inside but it requires us to write the members of those other implemented interfaces.

The *IEnumerable* interface has to be implemented to assure that the result of the query can be enumerated to iterate through the collection.
Those are the definition of the generic type *IEnumerable* interface and its typed overload:

```
1 public interface IEnumerable<T> : IEnumerable
2     {
3         IEnumerator<T> GetEnumerator();
4     }
5
6 public interface IEnumerable : IEnumerable
7     {
8         IEnumerator GetEnumerator();
9     }
```

Inside the GetEnumerator() method appearing in both interfaces, the *Execute* method of the provider is called and the generated expression tree is then translated and evaluated. This is because the queries in LINQ are lazily evaluated, so they are executed only when the result is accessed.

The definition of the weakly typed IQueryable interface is:

```
1 public interface IQueryable : IEnumerable {
2         Type ElementType { get; }
3         Expression Expression { get; }
4         IQueryProvider Provider { get; }
5     }
```

Inside the IQueryable interface are just 3 read-only properties:

- ElementType: it represents the type of the elements that are returned when the tree is evaluated; in case of IQueryable<T> it returns the type T, in case of the weakly typed *IQueryable* interface it returns the actual type of the result;

- Expression: it represents the entire expression tree generated from the query;
- Provider: it represents the provider associated with the data source.

To represent a sorting query, so enable the use of operators such OrderBy or OrderByDescending, also the IOrderedQuery<T> and IOrderedQuery must be implemented.

The definition are, as we can see, both empty:

```
1 public interface IOrderedQueryable<T> : IQueryable<T>, IEnumerable<T>,
      IOrderedQueryable, IQueryable, IEnumerable
2     {
3     }
4
5 public interface IOrderedQueryable : IQueryable, IEnumerable
6     {
7     }
```

More interesting is what is inside the IQueryProvider, which is the interface that really do all the work.
The definition is the following:

```
1  public interface IQueryProvider {
2         IQueryable CreateQuery(Expression expression);
3         IQueryable<TElement> CreateQuery<TElement>(Expression expression);
4         object Execute(Expression expression);
5         TResult Execute<TResult>(Expression expression);
6     }
```

The *CreatQuery* method creates, based on the chosen overload, an IQueryable or an IQueryable<T> object, to evaluate the query represented by the given expression tree. It's used to create new IQueryable objects that stay associated with a specific provider.

The *Execute* method it's the core of a provider. It consumes an expression tree and executes it against the wished data source. Inside this method it is possible to do everything with the expression tree: translate it, optimize it and then execute it to obtain the results.

6.1.2 Implementation of the skeleton of the provider

To create the basic skeleton of our provider we can use the one written and presented by Matt Warren from Microsoft on his blog (http://blogs.msdn.com/mattwar/pages/linq-links.aspx). On the first Part of the series, the C# code for implementing an empty provider is shown. Everything is well explained by Matt himself then any further comment will be redundant.

The work presented in this thesis will fill the *Execute* method of the IQueryProvider interface in order to generate and execute the optimized SQL code.

6.2 Generating the Data Context

Another important step is the mapping of the entities of our database into the object environment. As stated earlier, LINQ only works on data mapped into objects. Since we will create a LINQ to SQL provider to query a database we have to map the structure of the queried database into the object environment. This will provide LINQ the data context of the query.

The data context is a class containing a method for each entities of the database, where its information can be retrieved. For example, for a table we have information about the name and the type of the attributes and its constrains. This make LINQ to SQL a *strongly typed* C# code because the compiler knows at compile time the types of the queried database entities giving back errors not at runtime but at compile time.

Normally this work is done by the Object Relational Designer, that offers a visual design tool to map the objects in the database. It creates out of it the needed classes and generates a strongly-typed data context used to send and receive data between the entity classes and the database. But this generated data context is binded to the Microsoft LINQ to SQL provider and can't be changed.

There are other methods to create this data context, by writing by hand this data context, by using the Sqlmetal tool (ref to msdn) or to write a *template*.

Doing it by hand is out of question. Every time we want to use the provider to query a different database, this data context must be modified, requiring a lot of work.
Using the SqlMetal tool will do that for us, but also that doesn't allow us to substitute the LINQ to SQL provider during the generation of the data context.

A *template* can be used to generate dynamically the classes that will encapsulate our database.

In short, a template is a mechanism to generate functions and classes based on type parameters. They are used when the same algorithm has to be applied to different types of data. To avoid the creation of a separate class for each type containing the same code, the types are declared generics and, when an object is instantiated, substituted with the right type.

In our case we have to create a class containing all the information for each entities contained in our database. The structure of those classes will be always the same, so a template is the right solution.

To do that the template provided by http://www.codeplex.com/l2st4 can be re-used. By following the guide on the website a data context for the chosen database can be easily generated. This generated data context is however set to use the Microsoft LINQ-to-SQL provider.

In order to do generate a data context that will use our provider, the main file of the template (the one with .tt extension) needs to be modified.

First this method needs to be added to our generated data context:

```
1 MyQueryable<T> GetMyQueryable<T>(DataContext LINQDataContext) where T :
      class
2         {
3             MyQueryable<T> queryable = new MyQueryable<T>(new
                 MyQueryProvider());
4             queryable.setDataContext(this);
5             queryable.setLINQDataContext(LINQDataContext);
6             return queryable;
7         }
```

where MyQueryable is the name of our class implementing the IQueryable<T> interface.

In this method a new instance of our *IQueryable* implementation is instantiated and returned. The data context inside this object are set with the informations about the entity of the database that is accessed.

In our case this should happen when the information of a table are retrieved. In order to do that, inside the loop that generate the classes with the table infos, we need to add a call to our method and return this generated object with the right data context. Of course also the return type of the method should be changed.

The actual code looks like:

```
1 <\# foreach(Table table in data.Tables) {
2 \#>    <\#=code.GetAccess(table.BaseClass.TypeAttributes)\#>Table<<\#=table.
      BaseClass.QualifiedName\#>> <\#=table.Member\#>
3     {
4         get { return GetTable<<\#=table.BaseClass.QualifiedName\#>>(); }
5     }
6 <\# }
```

The modified code will be:

```
1  <\# foreach(Table table in data.Tables) {
2  \#>    <\#=code.GetAccess(table.BaseClass.TypeAttributes)\#>MyQueryable<<\#=
       table.BaseClass.QualifiedName\#>> <\#=table.Member\#>
3      {
4         get
5         {
6              Table<<\#=table.BaseClass.QualifiedName\#>> table = GetTable<<\#=
                   table.BaseClass.QualifiedName\#>>(); // hack to load all the
                   metamodelinfos...
7                  return GetMyQueryable<<\#=table.BaseClass.QualifiedName\#>>(
                       table.Context);
8         }
9      }
10 <\# }
```

To disambiguate it from the actual LINQ data context class (which is called DataContext) a new name should be given to this generated class. The name can be set by replacing all the occurrence of *<#=data.ContextName#>* in the template with a new chosen name, for example with *MyDataContext*.

To use our provider, in the application we just have to instantiate, instead of a DataContext object, our generated data context. Then all the queries should be done against it.

Chapter 7

Conclusions

The LINQ technology represents a big step in the usability of programming languages. Being able to query different data sources using the same syntax allows developers to write much more readable code, do operations that were total unimaginable before it, like use different data sources inside the same query, and extend it for user-specific uses.

But, as with every new technology, the initial approach can be in some cases immature, and the chosen method can be not the best one.
In particular the LINQ-to-SQL provider, used to handle the database data sources, uses an approach which is *data driven*, meaning that the number of generated queries is proportional to the queried data, and therefore all their intermediate results are put inside the heap and there processed. This method is fast when the query is simple and the queried data is small. But when the query starts to be a little bit more complex (having for example a nested query) and the data's size increases, this method's performances start to deteriorate, leading, in the worst case, to an error due to heap failure because the evaluation of the inner query is done for every tuple of the outer query and all those intermediate results are kept and processed inside the heap.

This proposed approach is instead *type driven*, meaning that the number of generated SQL queries depends only on the form of the result the user wants and not on the size of the queried data. This leads to move all the work inside the DBMS, which is heavily optimized to perform that type of work, instead of retrieving data from the database and process it inside the heap.
The generated SQL queries will be of course more complex and so need more time to be executed. For easy queries on small data the original approach is faster, but when the complexity of the queries and the size of the data increase this approach leads to far better performances,

terminating also many queries that the original provider can't handle.

The running example's query showed in section 3.6.1 generates, when executed using the Microsoft LINQ-to-SQL provider on an instance of the TPC-H database containing approx 1 Mb of data, 668 SQL queries and therefore opens also 668 connection towards the database. Using this new provider only the 4 queries showed in section 5.4.1 are generated, independently of the size of the database. It's important to notice that maybe not every query will be executed against the database, reducing even more the resourced used. It depends on what the end user will access.

The strict ordering introduced by the *loop-lifting* technique allows also the use of positional operators such as ElementAt not handled by the current LINQ-to-SQL provider. Also the Concat operator doesn't have any limitation based on the form of the input sequences because each object type is flattened as a relational table. In table 7.1 all the operators that the provider can handle are shown. The approach to handle some of them is however not shown in this thesis.

Another advantage of this approach is that the generate SQL code, contrary to the Microsoft provider, can be executed on any SQL:1999 capable DBMS because it doesn't depend, like the Microsoft provider, on Microsoft SQL Server behavior.

This proposed approach can be improved by the introduction of relative position instead of absolute position in the compilation phase. Maintain absolute position is not mandatory and really costly due to the introduction of many row numbering operators that needs all the data ready before they can be applied. Absolute position are required only by operators that works on positions, such as ElementAt or Take. A better approach is to modify all the compilation rules and instead of calculate a new position column at end of each rule, calculate a new absolute position column inside rules that really need it or when no other column can be used as relative position column.

This thesis ends with the query result inside the heap. To end its journey, like in the original LINQ-to-SQL provider, the result must be mapped back inside the object structure defined by the user in the query. This is another big research topic, because the object type must be defined at run time and, when the query result is ready in the heap, objects to contain it must

LINQ SQOs over `IQueryable<t>`	**New LINQ Provider**	**Microsoft LINQ Provider**
`Aggregate`	✓	✓
`All`, `Any`	✓	✓
`Average`, `Count`, `Sum`	✓	✓
`Concat`	✓	(unordered)
`Contains`	✓	✓
`Distinct`	✓	✓
`ElementAt`	✓	×
`Except`, `Intersect`, `Union`	✓	✓
`First`	✓	(unordered)
`GroupBy`, `GroupJoin`	✓	✓
`Join`	✓	✓
`Last`	✓	×
`Max`, `Min`	✓	✓
`OrderBy`, `ThenBy`	✓	✓
`Reverse`	✓	×
`Select(`v => . . . `)`	✓	✓
`Select((`v`,`p`)` => . . . `)`	✓	×
`SelectMany(`v => . . . `)`	✓	✓
`SelectMany((`v`,`p`)` => . . . `)`	✓	×
`SequenceEqual`	✓	✓
`Single`	✓	✓
`Skip`	✓	(unordered)
`SkipWhile`	✓	×
`Take`	✓	(unordered)
`TakeWhile`	✓	×
`Where(`v => . . . `)`	✓	✓
`Zip`	✓	×

Table 7.1: Support for the LINQ family of SQOs. A check mark (✓) indicates faithful support of the LINQ semantics, including order.

be instantiated. A way to prevent the instantiation of maybe million of objects at once has to be found to prevent the heap to collapse.

Bibliography

[1] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Pathfinder: XQuery-The Relational Way. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1322–1325, 2005. 2, 9, 95

[2] Torsten Grust, Manuel Mayr, Jan Rittinger, Sherif Sakr, and Jens Teubner. A SQL:1999 Code Generator for the Pathfinder XQuery Compiler. In *SIGMOD Conference*, pages 1162–1164, 2007. 2, 9, 95

[3] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: Database-supported program execution. In *SIGMOD international conference on Management of data*, 2009. 2, 46

[4] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proc. of the 30th International Conference on Very Large Databases (VLDB)*, pages 252–263, 2004. 2, 72

[5] Torsten Grust and Jens Teubner. Relational Algebra: Mother Tongue – XQuery: Fluent. In *Proc. of the 1st Twente Data Management Workshop on XML Databases and Information Retrieval (TDM)*, pages 7–14, 2004. 2, 72

[6] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the .net framework. In *SIGMOD international conference on Management of data*, 2006. 2

[7] Jan Rittinger, Jens Teubner, and Torsten Grust. Pathfinder: A relational query optimizer explores xquery terrain. In *GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web*, 2007. 2, 9, 95

[8] H. J. Schek and M. H. Schol. The relational model with relation-valued attributes. *Information Systems archive Volume 11, Issue 2*, pages 137 – 147, 1986. 3

[9] Tom Schreiber. Übersetzung von list comprehensions für relationale datenbanksysteme. Master's thesis, Technische Universität München, 2008. 2, 46, 77