

Thirteen New Players in the Team: A Ferry-based LINQ to SQL Provider

Tom Schreiber Simone Bonetti Torsten Grust Manuel Mayr Jan Rittinger

WSI, Universität Tübingen
Tübingen, Germany

`<firstname.lastname>@uni-tuebingen.de`

ABSTRACT

We demonstrate an efficient LINQ to SQL provider and its significant impact on the runtime performance of LINQ programs that process large data volumes. This alternative provider is based on FERRY, compilation technology that lets relational database systems participate in the evaluation of first-order functional programs over nested, ordered data structures. The FERRY-based provider seamlessly hooks into the .NET LINQ framework and generates SQL code that strictly adheres to the semantics of the LINQ data model. FERRY comes with strong code size guarantees and complete support for the LINQ Standard Query Operator family, enabling a truly interactive and compelling LINQ demonstration. A variety of inspection holes may be opened to learn about the internals of the FERRY-based LINQ to SQL provider.

1. SINGLE-LINE CHANGE—BIG IMPACT

“Will we be able to defeat the Bs this time?” This can be a pressing question if you are the coach of team A. You peer at the large `players` table of this year’s Basketball season (Table 1), featuring the player data of all teams. A list of the individual team rosters and a classification of a team’s players by their position—center, forward, or guard—would be more helpful now. You reach for your laptop computer to start the C# development environment. A brief program, formulated using the .NET Framework’s Language Integrated Query facility (LINQ), will do the job. You quickly create a *data context*, a LINQ abstraction that represents (1) a connection to a relational database back-end as well as (2) selected tables hosted by this back-end (including table `players`).

players			
team	name	pos	eff
:	:	:	:
A	Aaron	C	30
A	Andor	C	33
A	Arndt	F	20
A	Allan	F	22
A	Andre	F	21
A	Artur	G	15
A	Anton	G	16
:	:	:	:
B	Benny	C	35
B	Bobby	F	19
B	Boris	F	24
B	Brian	G	18
B	Bruno	G	23
B	Baldo	G	16
:	:	:	:

Table 1: Players.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.

Proceedings of the VLDB Endowment, Vol. 3, No. 2
Copyright 2010 VLDB Endowment 2150-8097/10/09... \$ 10.00.

```
1 var rosters =
2   from p in db.players
3   group p by p.team into team
4   let posn = from tp in team
5               group new { name = tp.name,
6                           eff = tp.eff } by tp.pos
7
8   select new {
9     team      = team.Key,
10    centers    = Single(Where(posn, pos => pos.Key == "C"))
11    forwards   = Single(Where(posn, pos => pos.Key == "F"))
12    guards    = Single(Where(posn, pos => pos.Key == "G")) };
```

Figure 1: The coach’s program: Compute team rosters, classify players by position, record player names and efficiency.

Once the data context is assigned to variable `db`, you enter the LINQ program of Figure 1.¹ The program groups players by team and partitions its players according to the position they assume within the team (encoded by C, F, G in column `pos`). Since this program operates over relational data, the C# runtime relies on the LINQ to SQL *provider* in charge to transform the query expression of lines 2–12 into a sequence of SQL statements.

As you execute the program, the hard disk starts grinding. You wait—and then check the query log of the connected database back-end: a continuous flood of SQL query statements, initiated by the LINQ to SQL provider, hits the database. This will not end in time... The match’s first quarter is already close as you interrupt execution and turn back to the C# editor. A *single-line change* to the data context creation effects a switch from the .NET-supplied LINQ to SQL provider to the alternative FERRY-based LINQ to SQL provider. You compile and execute again. The query log reports the execution of exactly four SQL queries before the program’s result (Figure 3) is printed instantly. With the rosters available in this form, it is straightforward to plan a lineup of A that can effectively confront team B—Section 3 will show how.

You close the lid of your laptop and watch team A win the match with a buzzer beater.

2. THE FERRY LINQ to SQL PROVIDER

A LINQ to SQL *provider* implements a SQL code generation and evaluation facility that is invoked whenever a C# (or VisualBasic) host program starts to enumerate the result of an embedded LINQ query expression over relational

¹Instead of the customary $e_1.f(e_2, \dots, e_n)$, we use C#’s static method syntax $c.f(e_1, e_2, \dots, e_n)$ to invoke the extension method f . For brevity, we omit the c . class prefix.

League Size	.NET LINQ		FERRY-based LINQ		
	# teams	# queries	 (sec)	# queries	 (sec)
100	601		0.406	4	0.140
1 000	6 001		20.265	4	0.438
10 000	60 001	> 30 mins		4	3.200
100 000	600 001	DNF		4	33.810

Table 2: Number of SQL queries emitted by the LINQ to SQL providers in dependence of the league size (20 players/team on average) and observed wall-clock execution times (average of 10 runs; DNF: did not finish within 16 hours).

data. The provider compiles the embedded LINQ query—represented in terms of a chain of *Standard Query Operator* (SQO) invocations—into a sequence of SQL statements that jointly implement the LINQ query’s semantics. These providers thus are instrumental in realizing the potential of the Language Integrated Query idea. LINQ is widely perceived as a major step towards a true integration of database and programming languages [1].

We demonstrate a new LINQ to SQL provider that is based on FERRY, query compilation technology that has been specifically designed to let relational database systems support the evaluation of first-order functional programs over ordered, nested list data structures. FERRY uses an algebraic compilation strategy, coined *loop lifting* [3], to translate the side-effect-free iteration embodied by LINQ’s central *from-in* comprehension construct [2, 6]. The absence of side effects yields a critical amount of *independent work* that loop lifting can turn into efficient bulk operations. Any SQL:1999-capable back-end may execute the resulting queries—FERRY’s code generator is not tied to SQL Server, in particular [3].

Query avalanche safety. The switch from the .NET-supplied LINQ to SQL provider to its FERRY-based variant brings with it a number of user-facing enhancements and extensions (Section 3). More fundamentally, we further obtain the guarantee that the *number of SQL queries issued* to implement a given LINQ query expression is exclusively *determined by the expression’s static type*: each occurrence of a list type constructor $[t]$ accounts for exactly one SQL query [4]. The query of Figure 1 and its result type of shape $[[\cdot]\cdot][\cdot]^2$, thus led to the bundle of four SQL queries which the coach observed in Section 1.

This marks a significant deviation from the LINQ to SQL provider that comes enclosed with .NET: here, the length of the SQL statement sequence can reflect the *database instance size*. (Note that .NET LINQ to SQL’s *deferred loading* feature makes no difference for the coach’s single-table query.) The ultimate result is an avalanche of queries that may very well overwhelm the relational database back-end (Section 1)—one example of the infamous *1+n Query Problem* experienced by many object-relational mappers.

FERRY’s approach to LINQ compilation gains a significant performance benefit. Table 2 reports on the SQL statement execution times for varying database instance sizes (*i.e.*, number of teams in the league), recorded on a contemporary Intel Core 2 Duo computer hosting .NET Framework 4. The avalanche effect quickly results in an overall query workload

²In this type shape, we consider list type constructors $[t]$ (abbreviating the C# type `IQueryable<t>`) only.

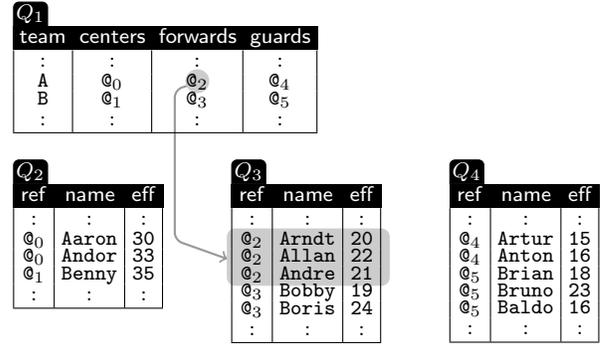


Figure 2: Result of the four-query bundle Q_{1-4} : tabular encoding of a nested list of type $[[\cdot][\cdot][\cdot]]$.

```
[ ...,
  { team = "A", centers = [ { name = "Aaron", eff = 30 },
                          { name = "Andor", eff = 33 } ],
    forwards = [ { name = "Arndt", eff = 20 },
                 { name = "Allan", eff = 22 },
                 { name = "Andre", eff = 21 } ],
    guards = [ { name = "Artur", eff = 15 },
               { name = "Anton", eff = 16 } ] },
  ...,
  { team = "B", ... },
  ... ]
```

Figure 3: Excerpt of the nested result of the coach’s program, bound to the C# variable `rosters`.

that prohibits timely completion, let alone interactive use.

Query bundles and partial execution. FERRY employs a non-parametric LINQ data model encoding that (1) uses an inline representation for atomic values and (2) relies on foreign surrogate keys—much like NF^2 databases [8]—to represent embedded lists. The four-query bundle of Section 1, for example, computes four tables Q_{1-4} which collectively encode the coach’s query result (Figures 2 and 3). A bundle’s queries are independent and may be evaluated in any order or even concurrently. This is contrast to the just mentioned avalanches whose queries are executed in a serial, iterative fashion [4]. The independence of its constituent queries further enables the partial execution of a bundle. Consider a LINQ host program that projects variable `rosters` onto a value of type $[[\cdot]]$:

```
1 foreach (var r in rosters) {
2     Console.WriteLine("Team {0} has {1} forwards.",
3                       r.team, Count(r.forwards));
}
```

With the FERRY-based LINQ to SQL provider in place, the C# runtime evaluates the queries Q_1 and Q_3 only—the vanilla .NET provider still submits the entire query avalanche for execution.

3. THIRTEEN NEW SQOS AND BEYOND

The FERRY-based provider has been designed to strictly play by the rules of the LINQ semantics, enabling a new *bona fide* LINQ to SQL experience.

Preservation of list order. List order is inherent to the LINQ data model [2] and FERRY’s compiler derives and, where required, propagates a query-accessible representation of order [3] when it processes base tables like `players`. Order preservation (1) brings into reach a group of order-sensitive

```

1 var homeFwds =          var awayFwds =
2 Take(from r in rosters  Take(from r in rosters
3   where r.team == "A"   where r.team == "B"
4   from p in r.forwards  from p in r.forwards
5   order by p.eff descending order by p.eff descending
6   select p),            select p),
7   n);                  n);

```

Figure 4: The n most efficient forwards of teams A and B.

```

[ { h = { name = "Allan", eff = 22 },
  a = { name = "Boris", eff = 24 } },
  { h = { name = "Andre", eff = 21 },
    a = { name = "Bobby", eff = 19 } } ]

```

Figure 5: A lineup reflecting relative player efficiency, bound to C# variable lineup.

SQOs that received no or only partial support before, and (2) establishes basic laws like

$$\text{Concat}(\text{Take}(e, n), \text{Skip}(e, n)) = e, \quad (*)$$

that make order-dependent LINQ programs practical. Without a runtime representation of list order, the .NET-supplied LINQ to SQL implementations of the order-sensitive SQOs operate based on *some* row ordering prescribed by the relational back-end and cannot guarantee (*).

To illustrate the value of order in the LINQ data model, consider a lineup of the forwards of teams A and B in which players of the same relative efficiency (eff) shall oppose each other. Given the ordered lists containing the most efficient forwards of both teams (see `homeFwds`, `awayFwds` in Figure 4), the lineup is quickly computed by a `Zip`, a positional join that moves a “slider” function across two lists in synchronization:

```

1 var lineup =
2 Zip(homeFwds, awayFwds, (h, a) => new { h, a });

```

Figure 5 shows the resulting lineup in which the four best forwards confront as desired. An attempt to simulate this use of the order-sensitive `Zip` SQO in absence of FERRY’s built-in support quickly turns the above one-liner into an unwieldy composition of `Join` with nested `Selects`.

Order preservation on a relational back-end does not come for free and FERRY’s algebraic compiler and optimizer go a long way to derive that the semantics of a given query (sub-)expression does *not* depend on order [5]. In such cases, order derivation and propagation effort will be judiciously removed from the issued SQL query code.

Faithful support for the SQO family. FERRY faithfully implements LINQ’s order-sensitive operators, *e.g.*, `ElementAt`, `Reverse`, positional mapping via `Select((v,p)=>...)`, or `Zip` as introduced with .NET 4.0 (see Table 3). These thirteen SQOs now join the rest of the family of admissible operators in LINQ to SQL programs. This makes FERRY’s support complete for all SQOs applicable to objects of type `IQueryable<t>`. In addition to the `IQueryable` SQOs, FERRY can embrace the `IEnumerable` interface, most notably `ToLookup` and the related associative indexing operation $e_1 [e_2]$, which allow a particularly elegant and even shorter formulation of the coach’s query (Figure 6).

Loop lifting, an efficient non-parametric value representation, and order preservation open the door towards a considerably richer set of built-ins beyond the Standard Query Operators defined by LINQ. Examples include a versatile

FERRY	Standard Query Operators	.NET
faithful	Aggregate, All, Any, Average, Count, Contains, DefaultIfEmpty, Distinct, Except, Intersect, GroupBy, GroupJoin, Join, Max, Min, OrderBy[Descending], ThenBy[Descending], Select($v=>...$), SelectMany($v=>...$), SequenceEqual, Single[OrDefault], Sum, Where($v=>...$),	faithful
	Concat, First[OrDefault], Skip, Take, Union,	
	ElementAt[OrDefault], Last[OrDefault], Reverse, Select($(v,p)=>...$), SelectMany($(v,p)=>...$), SkipWhile, TakeWhile, Where($(v,p)=>...$), Zip, Empty*, Range*, Repeat*, ToLookup*	none

Table 3: Levels of database support for the SQO family in the FERRY-based and .NET-supplied LINQ providers (SQOs marked with * operate on objects of type `IEnumerable<t>`).

```

1 var rosters =
2 from p in db.players
3 group p by p.team into team
4 let posn = ToLookup(team, tp => tp.pos,
5                   tp => new { name = tp.name,
6                             eff = tp.eff })
7
8 select new {
9   team = team.Key,
10  centers = posn["C"],
11  forwards = posn["F"],
12  guards = posn["G"] };

```

Figure 6: A variant of the coach’s query, using the `ToLookup` SQO supported by the FERRY LINQ to SQL provider.

list-based grouping primitive inspired by Haskell’s list comprehensions [3, 6], `Zip`’s complement `UnZip`, or the (local) announcement of order indifference (`Unordered`, [5]). Query expressions written in such an extended dialect of LINQ come close to list-processing programs written in first-order functional programming languages.

4. DEMONSTRATION SETUP

The demonstration (Figure 7) will come packaged with both, authentic National Basketball Association (NBA) relational box score and player data as well as TPC-H benchmark instances of varying scale factors. A series of C# programs plus embedded LINQ fragments are prepared to explore these data sets.

The FERRY-based LINQ to SQL provider has been constructed as a drop-in replacement for the query functionality of the .NET-supplied original.³ A single-line change to the data context declaration suffices to make the demonstration switch from the .NET-enclosed provider to the FERRY variant and back (recall Section 1 and see Figure 8).

Users may challenge the FERRY provider via ad-hoc LINQ query expressions that will be compiled and executed on the click of a button (the demonstration is based on the Microsoft[®] Visual Studio .NET development environment). With FERRY’s implementation of the SQO family and support for further list-processing primitives, the demonstration can cope

³LINQ’s update tracking has not yet been incorporated into the FERRY-based provider.

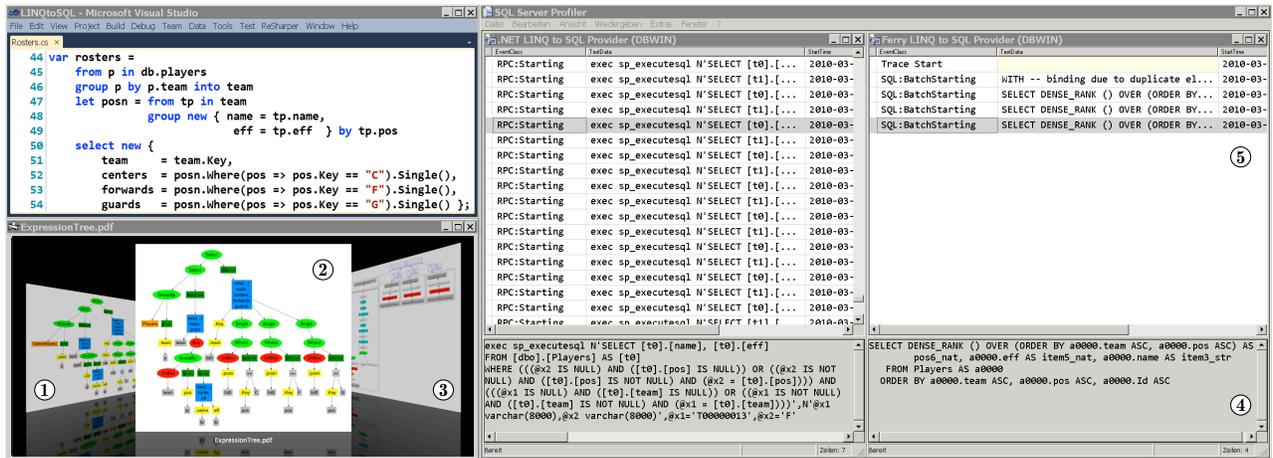


Figure 7: Demonstration setup: C# editor, rendered LINQ expression trees and graphical representation of algebraic query plan bundles, SQL query logs and code (FERRY on the right-hand side). Numbers in \bigcirc refer to the provider stages of Figure 9.

```

41 var db = new DataContext(connection);
42
43
44 var rosters =
45     from p in db.players
46     group p by p.team into team

```

(a) Switching between the .NET LINQ provider and ...

```

41 var db = new Ferry.DataContext(connection);
42
43
44 var rosters =
45     from p in db.players
46     group p by p.team into team

```

(b) ...the FERRY-based LINQ to SQL provider requires a single line of change (here: insert/delete the *Ferry* namespace prefix in line 42).

Figure 8: The FERRY-based LINQ provider hooks seamlessly into the .NET framework. During the demonstration, the provider in charge may be toggled arbitrarily.

- (1) with “relational style” programs that extract, connect, filter, group, and aggregate tabular data, and
- (2) with programs that follow a compositional first-order⁴ “functional style” to process ordered—and potentially deeply nested—lists.

A mix of both styles, allowing complex yet compact queries over relational source data, leads to a unique and compelling experience of writing data-intensive programs.

Under the hood, the FERRY provider is organized in a series of stages (Figure 9). In preparation of SQL code generation, the provider relies on an extensive data flow analysis to simplify and (drastically) reshape the initial algebraic query bundles. To learn about these internals, the demonstration opens a number of inspection holes (Figure 7) that give insight into FERRY’s algebraic compilation strategy, loop lifting in particular. Users may flip through the rendered output of all relevant stages, e.g., LINQ expression trees or algebraic query plan bundles before and after optimization.

⁴As we write this, FERRY is extended to support closures and higher-order functions [7].

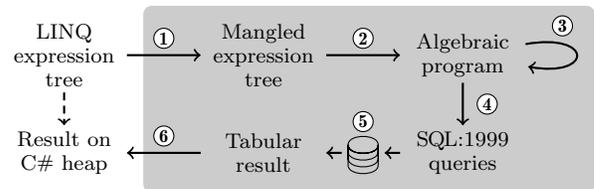


Figure 9: Stages in the FERRY-based LINQ provider.

Finally, the demonstration setup incorporates (extracts of) the back-end’s SQL query logs. A peek into these logs—just like team A’s coach did in Section 1—manifests the significant impact of FERRY’s query avalanche safety feature.

Acknowledgments. This research has been supported by the German Research Foundation (DFG), Grant GR 2036/3-1.

Additional information on FERRY is available on the Web at www.ferry-lang.org.

5. REFERENCES

- [1] R. Agrawal et al. The Claremont Report on Database Research. *CACM*, 52(6), 2009.
- [2] G. M. Bierman, E. Meijer, and M. Torgersen. Lost In Translation: Formalizing Proposed Extensions to C#. In *Proc. OOPSLA*, 2007.
- [3] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: Database-Supported Program Execution. In *Proc. SIGMOD*, 2009.
- [4] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-Safe LINQ Compilation. In *Proc. VLDB*, 2010.
- [5] T. Grust, J. Rittinger, and J. Teubner. eXrQuery: Order Indifference in XQuery. In *Proc. ICDE*, 2007.
- [6] S. P. Jones and P. Wadler. Comprehensive Comprehensions: Comprehensions with “Order by” and “Group by”. In *Proc. Haskell Workshop*, 2007.
- [7] J. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation*, 11(4), 1998.
- [8] H. J. Schek and M. H. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 11(2), 1986.