Avalanche-Safe LINQ Compilation

Torsten Grust Jan Rittinger Tom Schreiber

WSI, Universität Tübingen
Tübingen, Germany

torsten.grust | jan.rittinger | tom.schreiber@uni-tuebingen.de

ABSTRACT

We report on a query compilation technique that enables the construction of alternative efficient query providers for Microsoft's Language Integrated Query (LINQ) framework. LINQ programs are mapped into an intermediate algebraic form, suitable for execution on any SQL:1999-capable relational database system.

This compilation technique leads to query providers that (1) faithfully preserve list order and nesting, both being core features of the LINQ data model, (2) support the complete family of LINQ's Standard Query Operators, (3) bring database support to LINQ to XML where the original provider performs in-memory query evaluation, and, most importantly, (4) emit SQL statement sequences whose size is only determined by the input query's result *type* (and thus independent of the database size).

A sample query scenario uses this LINQ provider to marry database-resident TPC-H and XMark data—resulting in a unique query experience that exhibits quite promising performance characteristics, especially for large data instances.

1. LINQ QUERY PROVIDERS

Language Integrated Query (LINQ) seamlessly embeds a declarative query language facility into Microsoft's .NET language framework [14]. Developers use the familiar idioms of their programming language—say, C#—plus a generic set of *Standard Query Operators* (SQOs) to formulate queries against lists of heap-resident C# objects, XML trees, or relational tables. LINQ *providers* then translate these programs into an executable form chosen to match the kind of queried data: a LINQ query against database-resident relational tables is compiled into a sequence of SQL statements, for example. This translation is performed by the framework's LINQ to SQL provider. LINQ is widely perceived as a signpost to a new declarative form of data access [2, 21] and the availability of efficient LINQ providers will be instrumental.

The principal LINQ construct is the *query comprehension*, a generic iteration facility applicable to any data type

Proceedings of the VLDB Endowment, Vol. 3, No. 1


```
1 XElement.Load("auction.xml").Descendants("closed_auction")
2   .SelectMany(auction => db.GetTable<Lineitem>(),
3                          (auction, li) => new {auction, li})
4   .Where(t => (int)t.auction.Element("itemref")
5                            .Attribute("item") == t.li.partkey)
6   .GroupBy(t => t.li.orderkey,
7            t => new { partkey = t.li.partkey,
8                       diff    = t.li.extendedprice -
9                         (decimal)t.auction.Element("price") })
10  .Select(g => new { order    = g.Key,
11                     gain     = g.Sum(t => t.diff),
12                     partkeys = g.Select(t => t.partkey) })
```

Figure 1: Program P, rendered as a serialized LINQ expression tree. An equivalent in user-facing query syntax is shown in Paragraph 1 (Appendix).

that exhibits the properties of a monad [25] (in LINQ, this role is assumed by the generic types `IQueryable<t>` and `IEnumerable<t>`). In a single LINQ query, developers may mix and match operations against XML documents as well as relational tables and then rely on the LINQ providers in charge to perform the required data-specific operations under the hood.

As of today, this conceptual appeal of the LINQ approach is not fully realized by the LINQ provider implementations that come enclosed with the .NET framework. While the current LINQ to SQL and LINQ to Entities providers emit SQL statements and thus delegate query evaluation to a back-end RDBMS, all operations performed by the LINQ to XML provider are performed in-memory (on heap-resident representations of XML nodes). This mismatch of execution sites leads to asymmetries: in a nested LINQ query comprehension, the provider in charge of the *outermost* enclosing iteration "takes the lead."

The sample Program P in Figure 1 features a typical scenario of nested iteration: the leading `SelectMany` in line 2 iteratively binds variable `auction` to `closed_auction` element nodes extracted from XML document `auction.xml` (see Paragraph 1 (Appendix) for comments on Program P and a sketch of the queried data). The LINQ to XML provider thus is in charge of the outer iteration. Each iteration leads to an invocation of the LINQ to SQL provider which is in charge of the inner iteration over table `Lineitem`, binding rows to variable `li` in line 3. This mode of evaluation hits the RDBMS with an avalanche of SQL queries whose results have to be transferred into the heap for further evaluation by LINQ to XML (given that main memory capacity suffices). In the case of Program P, the number of `closed_auction` XML elements determines the size of that query avalanche, amounting to 9 750 SQL queries for an `auction.xml` XMark document of scale factor 1.0, for example.

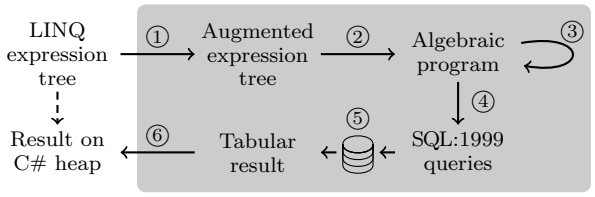


Figure 2: Stages in the Ferry-based LINQ provider.

A Ferry-based LINQ query provider. In the present work, we describe an approach to LINQ provider construction that is safe from query avalanches. Under the new regime,

> *it is exclusively a LINQ query's static result* type*—not the database instance size—that determines the number of initiated database queries.*

This marks a significant deviation from the .NET-supplied providers. The performance impact is profound (Section 6). The query avalanche issue affects programs that rely on an interaction of the LINQ to Entities [1] or LINQ to SQL providers with LINQ to XML, but also LINQ to SQL-only programs may lead to the phenomenon (see Paragraph 6 (Appendix)).

One option to resolve the additional problem of execution site asymmetry (in-memory *vs.* by-database) is to shift XML processing into the database back-end. We thus

> *let the RDBMS host and process relational data as well as—potentially sizable, larger than main memory—XML instances,*

represented according to one of the common tabular or native formats [10, 16, 22].

The alternative query provider builds on Ferry [9], query compilation technology that enables off-the-shelf RDBMSs to participate in the evaluation of functional programs over nested, ordered list structures (a particularly good match for the LINQ data model). This enables

> *database support for the complete family of LINQ SQOs and faithful preservation of the semantics of list nesting and order.*

The Ferry-based provider employs a query compiler that emits a table algebra dialect carefully designed to mimic the capabilities of modern SQL:1999-capable database backends. All relevant Ferry pieces are covered by this paper—further details may be found in [7–9].

The resulting Ferry-based LINQ provider hooks into the .NET framework and can act as drop-in replacement of the existing provider's query functionality [20]. LINQ's update tracking is not addressed in the present work.

In what follows, we will walk Program P through the stages of the Ferry-based LINQ provider (Figure 2). This entails normalization (①, Section 2), a discussion of the faithful relational representation of LINQ's data model (Section 3), algebraic compilation (②, Section 4), as well as algebraic optimization, SQL code generation, and execution (③–⑥, Section 5). Section 6 assesses the significant performance impact of avalanche safety before we review closely related efforts (Section 7). Section 8 concludes the discussion.

2. LINQ'S COMPREHENSION CORE

The LINQ provider takes over after the C# compiler has transformed the user-facing LINQ query syntax of Figure 10

```
1 XElement.Load("auction.xml").Descendants("closed_auction")
2   .SelectMany(auction => table Lineitem (orderkey,...)
3                          with key (orderkey,linenumber),
4               (auction, li) => (auction, li))
5   .Where(t => (int)t.1.Element("itemref")
6                    .Attribute("item").Value == t.3)
7   .GroupBy(t => t.2,
8            t => (t.3,
9                  t.7 - (decimal)t.1.Element("price").Value))
10  .Select(g => (g.1,
11                g.2.UnBox().Sum(t => t.2),
12                g.2.UnBox().Select(t => t.1).Box()))
```

Figure 3: Serialized expression tree of Program P after augmentation and introduction of (Un)Box() invocations (see Section 3).

(Appendix) into an expression tree of chained SQO applications (Figure 1) [3]. Note how these linear SQO invocation chains bear an inherent asymmetry: the `SelectMany` early in the chain (line 2) iterates over the list of `closed_auction` elements and thus drives the whole query evaluation process. This leads to the query avalanche issues (iterative execution of a, potentially huge, number of SQL queries) outlined in Section 1.

As mentioned before, the LINQ semantics are built around a core iteration construct, the *query comprehension*. LINQ query comprehensions are a generalization of the list (or monad) comprehension concept [13]. Variants appear in a number of programming languages, *e.g.*, Perl 6, Python, Ruby, Scala, and, most notably, Haskell [11]. In LINQ, a comprehension takes the form

$$\texttt{from}\, v\, \texttt{in}\, e_1\, \texttt{select}\, e_2\ ,$$

or, equivalently, formulated using the `Select` SQO,

$$e_1\texttt{.Select (}v\texttt{ => }e_2\texttt{)}\ .$$

`Select` performs the iterated evaluation of expression e_2 under bindings of variable v—these bindings are generated from the values contained in e_1 of type `IQueryable<t>` (in other words: function v `=>` e_2 is mapped over e_1). In what follows, we will use the more compact $[t]$ to denote the type `IQueryable<t>` and write $[x_1, \ldots, x_\ell]$, with x_i of type t, to denote a list of type $[t]$. Similarly, we will write $(x_1, \ldots, x_n)$ to abbreviate the C# record constructed by `new {`f_1 `=` x_1`,...,`f_n `=` x_n`}` and use $(t_1, \ldots, t_n)$ to denote its anonymous type if x_i is of type t_i.

To prepare compilation, the Ferry-based LINQ provider applies simple normalizing rewrites to the expression tree, leading to the program of Figure 3 (ignore the Box() and UnBox() calls for now):

(1) database table references (`GetTable<`d`>()`) are replaced by explicit `table` d `(`$\cdots$`)` constructs, revealing table meta information like keys (lines 2, 3),
(2) C# `new {`$\cdots$`}` record constructors and field name references are turned into tuples and positional access (*e.g.*, `t.3` in line 6 replaces `t.li.partkey` in Figure 1),
(3) relationship traversals are turned into the appropriate table references, and
(4) XML node atomization is made explicit in terms of `Value` invocations (lines 6, 9).

We now discuss a suitable relational encoding of LINQ's ordered and nested data model before we describe the actual algebraic LINQ compiler in Section 4.

3. THE LINQ DATA MODEL ON A RELATIONAL BACK-END

This work aims to construct a database-supported LINQ query provider that properly reflects the semantics of LINQ's rich data model, featuring atomic types, XML node types, as well as list and tuple constructors (producing values of type $[t]$ and $(t, \ldots, t)$, respectively). The following relational representation of values of this data model enables a SQL database back-end to act as a LINQ processor that preserves this semantics (including list order and nesting).

Atomic values, XML nodes, tuples. LINQ queries may operate over values of atomic C# types. Just like the .NET LINQ to SQL provider, we use a direct mapping of such C# values into values of a corresponding SQL type[1], translating the C# type `decimal` into the SQL type `NUMERIC(28)`, for example. However, much *unlike* the .NET LINQ to XML provider, we employ ORDPATH-style hierarchical identifiers [16] to obtain a database-friendly representation of XML nodes. These identifiers encode a node's location in its containing XML document tree (thus facilitating XML document navigation) and are also used to perform lookups for further XML node properties—tag name, node kind, *etc.*—in table XMLdocs (Table 5, Appendix).

An n-tuple $(v_1, \ldots, v_n)$, $n \geqslant 1$, of such items maps into a table row of width n. A 1-tuple (v) and value v are treated alike.

Ordered lists. As the relational back-end itself does not provide row ordering guarantees, we let the compiler create a *runtime-accessible encoding of order*. A list value $[x_1, x_2, \ldots, x_\ell]$—let x_i denote the n-tuple $(v_{i1}, \ldots, v_{in})$—is mapped into a table of width $1 + n$ as shown in Figure 4(a). Again, a singleton list $[x]$ and its element x are represented alike. The attached pos column provides the hook required to faithfully implement the assorted order-sensitive LINQ SQOs, *e.g.*, the position-aware `Select((v,p) => ...)`, `ElementAt`, `First`, `Reverse`, `Skip`, or `TakeWhile`. .NET supplies a translation that, in general, cannot preserve the order semantics on the database back-end (see Section 4).

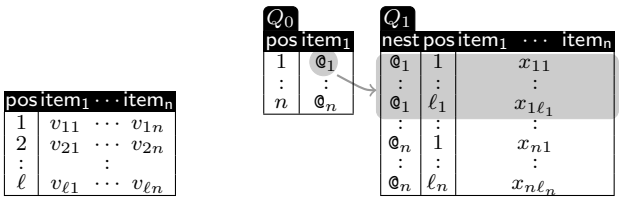
Nested lists. In the LINQ data model, tuple and list constructors may be nested to arbitrary depth. We embrace such nesting on the flat 1NF SQL database back-end via *surrogate* (foreign key) values, variants of which have been used to realize non-first normal form databases (NF^2) in the late 1980s [18].

If a LINQ program produces a nested value, say the list $[[x_{11}, x_{12}, \ldots, x_{1\ell_1}], \ldots, [x_{n1}, x_{n2}, \ldots, x_{n\ell_n}]]$, $n \geqslant 1$, the Ferry-based provider will fork the compilation process to translate the program into *a bundle* of two separate relational queries, Q_0 and Q_1. Figure 4(b) depicts the resulting tabular encodings produced by the relational query bundle:

- Q_0, one query that computes the relational encoding of the outer list $[\mathbb{0}_1, \ldots, \mathbb{0}_n]$ in which all inner lists (including empty lists) are represented by surrogates $\mathbb{0}_i$, and
- Q_1, one query that produces the encodings of *all* inner lists—assembled into a single table. If the ith inner list is empty, its surrogate $\mathbb{0}_i$ will not appear in the nest column of this table.

Note that the two constituent queries still are flat queries to be evaluated over an 1NF database. Emitting such *bundles*

[1] `http://msdn.microsoft.com/en-us/library/bb386947.aspx`.



(a) Encoding a flat ordered list.

(b) Encoding a nested list (Q_0: outer list, Q_1: all inner lists).

Figure 4: Relational runtime encoding of order and nesting on the database back-end.

of independent queries, like Q_0 and Q_1 above, enables the provider to selectively produce the query result at different list nesting levels—we briefly come back to this in Section 5.

In effect, the Ferry-based provider thus relies on a non-parametric representation of values [4] in which the *types* determine the efficient relational representation: in-line (for tuples of atomic items) *vs.* surrogate-based (for lists). Figures 5 and 6 define a compile-time analysis and annotation phase implementing the judgment $e = e' : \tau$, with $\tau \in \{atom, list\}$, to infer the representation required for normalized LINQ expression e.

(Un)Boxing. The annotation phase yields the augmented expression tree e' in which the `Box()` and `UnBox()` pseudo SQOs identify those sub-expressions that require a change of non-parametric value representation:

`Box()`, Rule 2 of Figure 5: If the result of a *list*-typed expression e_1 (or e_2) is to be embedded in a tuple, `Box()` instructs the compiler to fork the compilation process: a separate query is emitted that will compute the relational representation of e_1 (e_2). The resulting table will carry a nest column whose surrogates allow to refer to the nested results. Section 4 discusses the relational implementation of `Box()` and `UnBox()`.

`UnBox()`, Rule 8: If the method signature of SQO f indicates that the individual elements of a boxed argument list e_i $(2 \leqslant i \leqslant n)$ are required to compute f, `UnBox()` triggers the compiler to dereference the surrogates and emit a foreign key join to access the nested elements of e_i.

Figure 3 shows the expression tree of Program P after augmentation with `Box()` and `UnBox()`.

Types exclusively determine query forks. Observe that it is exclusively the *number of list constructors* $[\cdot]$ *in the query's result type* that determines the forks and thus number of queries contained in the emitted relational query bundle. For Program P and its result type

$$[(\texttt{int}, \texttt{decimal}, [\texttt{int}])] \ ,$$

the bundle size thus is 2 (peek forward at Figure 8 where the single fork is clearly visible). This is radically different from the .NET-enclosed LINQ to SQL provider which may yield sequences of SQL statements whose size is dependent on the *size of the queried database instance*, resulting in the severe query avalanche issue explained in Section 1. Under Ferry's particular approach to nesting, even LINQ programs yielding complex nested results will lead to a tractable and *statically predictable* number of queries. Section 5 quantifies the significant performance impact of this compilation technique.

$$\frac{e = e' : atom}{e.n = e'.n : atom}(1) \quad \frac{e_i = e_i' : \tau_i \big|_{i=1,2}}{(e_1, e_2) = (\boxplus^{\tau_1}_{atom}(e_1'), \boxplus^{\tau_2}_{atom}(e_2')) : atom}(2)$$

$$\frac{}{v = v : atom}(3) \quad \frac{e = e' : \tau}{(v_1, \ldots, v_n) \texttt{=>} e = (v_1, \ldots, v_n) \texttt{=>} e' : \tau}(4)$$

$$\frac{}{\begin{array}{l}\texttt{table}\, R\, (c_1, \ldots, c_n) \\ \texttt{with key}\, (k_1, \ldots, k_m)\end{array} = \begin{array}{l}\texttt{table}\, R\, (c_1, \ldots, c_n) \\ \texttt{with key}\, (k_1, \ldots, k_m)\end{array} : list}(5)$$

$$\frac{e = e' : atom}{(t)e = (t)e' : atom}(6) \quad \frac{e_i = e_i' : atom \big|_{i=1,2} \quad * \in \{\texttt{+}, \texttt{=}, \texttt{and}, \ldots\}}{e_1 * e_2 = e_1' * e_2' : atom}(7)$$

$$\frac{e_i = e_i' : \tau_i \big|_{i=1,2} \quad f :: (t_1, \ldots, t_n) \to t_{n+1}}{\begin{array}{l}e_1.f(e_2, \ldots, e_n) = \\ \boxplus^{\tau_1}_{\tau(t_1)}(e_1').f(\boxplus^{\tau_2}_{\tau(t_2)}(e_2'), \ldots, \boxplus^{\tau_n}_{\tau(t_n)}(e_n')) : \tau(t_{n+1})\end{array}}(8)$$

Figure 5: Inference of *atom*/*list* type annotations (non-parametric value representation).

$$\begin{array}{rcl}\tau([t]) & = & list \\ \tau((t_1, \ldots, t_n) \to t_{n+1}) & = & \tau(t_{n+1}) \\ \tau(t) & = & atom \\ \\ \boxplus^{\tau}_{\tau}(e) & = & e \\ \boxplus^{\sigma}_{\tau}((v_1, \ldots, v_n) \texttt{=>} e) & = & (v_1, \ldots, v_n) \texttt{=>} \boxplus^{\sigma}_{\tau}(e) \\ \boxplus^{list}_{atom}(e) & = & e\texttt{.Box()} \\ \boxplus^{atom}_{list}(e) & = & e\texttt{.UnBox()}\end{array}$$

Figure 6: $\boxplus^{\sigma}_{\tau}(e)$ introduces `Box()` or `UnBox()` if the actual (σ) and required (τ) non-parametric representations of e diverge.

4. AN ALGEBRAIC LINQ COMPILER

The inference rules of Figure 7 devise a compositional algebraic compilation scheme for LINQ. The compiler's target language is a table algebra (see Table 1) whose operators have been designed to match the capabilities of modern SQL:1999 query processors. Featured operators include base table access ($\textcircled{\scriptsize R}$), duplicate-preserving projection (π), selection (σ), cross product and join, ($\times$, $\bowtie$), and grouped aggregation (AGG). Duplicate rows are also preserved by $\cup$, $\setminus$ and are only eliminated by an explicit δ. Operators @, CAST, $\circledast$, and ϱ attach a new column to their input table. In particular, row ranking ϱ is used to correctly encode list order and to produce surrogates (see the discussion of columns pos and nest in Section 3): $\varrho_{\mathsf{a}:\langle \mathsf{b}_1, \ldots, \mathsf{b}_n \rangle / \mathsf{c}}$ attaches dense ranks $(1, 2, \ldots)$ inside each c-group in the order given by columns $\mathsf{b}_1, \ldots, \mathsf{b}_n$. Operator ϱ thus exactly mimics SQL:1999's `DENSE_RANK` function.

Operator	Semantics
$\pi_{\mathsf{a}_1:\mathsf{b}_1, \ldots, \mathsf{a}_n:\mathsf{b}_n}$	project onto columns b_i, rename b_i into a_i
σ_p	select rows satisfying predicate p
$\_ \times \_$	Cartesian product
$\_ \bowtie_{\mathsf{a}_1, \ldots, \mathsf{a}_n} \_$	natural join on columns $\mathsf{a}_1, \ldots, \mathsf{a}_n$
$\_ \bowtie_p \_$	join with predicate p
δ	eliminate duplicate rows
$\_ \cup \_$	disjoint union
$\_ \setminus \_$	difference
$@_{\mathsf{a}:v}$	attach constant value v in column a
$\mathrm{CAST}_{\mathsf{a}:(t)\mathsf{b}}$	attach value of b casted to type t in a
$\circledast_{\mathsf{a}:\langle \mathsf{b}_1, \ldots, \mathsf{b}_n \rangle}$	attach result of application $*(\mathsf{b}_1, \ldots, \mathsf{b}_n)$ in a
$\varrho_{\mathsf{a}:\langle \mathsf{b}_1, \ldots, \mathsf{b}_n \rangle / \mathsf{c}}$	group by c, attach row rank (in b_i order) in a
$\mathrm{AGG}_{\mathsf{a}:\langle \mathsf{b} \rangle / \mathsf{c}}$	group by c, compute aggregate of b in a
$\textcircled{\scriptsize R}_R$	read from database-resident table R
a b c – – –	literal table with columns a, b, c

Table 1: Intermediate table algebra (with n-ary operator $* \in \{\texttt{+}, \texttt{=}, \texttt{and}, \ldots\}$ and AGG $\in \{\mathrm{COUNT}, \mathrm{MAX}, \mathrm{MIN}, \ldots\}$).

Comprehensions and bulk-oriented evaluation. Relational query processors are specialists in *bulk-oriented evaluation*: in this mode of evaluation, the system applies an operation to *all* rows in a given table. In absence of inter-row dependencies, the system may process the individual rows in any order or even in parallel. To actually operate the database back-end in this bulk-oriented fashion, the compiler draws the necessary amount of independent work from LINQ's comprehensions. The query comprehension

$$[x_1, \ldots, x_n]\texttt{.Select}\,(v \texttt{=>} e) = [e\,[x_1/v], \ldots, e\,[x_n/v]]$$

performs n *independent* evaluations of expression e under different bindings of v ($e\,[x/y]$ denotes the consistent replacement of free occurrences of y in e by x). The compiler exploits these semantics and applies a translation technique, coined *loop lifting* [10], that compiles the comprehension into

an algebraic plan that receives the representation of $[x_1, \ldots, x_n]$ to produce a *single* table containing the representations of $e\,[x_i/v]$ $(i = 1, \ldots, n)$, *i.e.*, for *all iterations*.

A dedicated iter column in the produced table is used to tell the individual iterations apart. Loop lifting fully realizes the independence of the iterated evaluations and enables the relational query engine to take advantage of its bulk-oriented processing paradigm: the results of the individual evaluations of e may be produced in any order (or in parallel).

Here, we adopt and adapt loop lifting—which originally had been designed to compile XQuery's FLWOR construct [10], yet a different incarnation of (flat) comprehensions—to cope with LINQ's nested data model and the family of SQOs.

Loop lifting for LINQ. The rule set of Figure 7 collectively defines the "compiles to" ($\Rrightarrow$) relation via judgment

$$\Gamma; loop \vdash e \Rrightarrow (q, cs, ts)$$

which compiles LINQ expression e into the algebraic plan fragment q. The compiler maintains the following central invariant: if e is of type $[(t_1, \ldots, t_n)]$, plan q will produce a table with schema $\mathsf{iter}|\mathsf{pos}|cs$ (with $cs = \mathsf{item}_1|\ldots|\mathsf{item}_n$) of $2 + n$ columns in which

(1) columns iter, pos contain information about iteration and list order (see Section 3), and
(2) column item_i contains the values occurring in the ith tuple position. If t_i is a list type $[s_i]$, then item_i will be a column of surrogate values. In this case, the mapping ts will contain an entry $\mathsf{item}_i \mapsto (q_i, cs_i, ts_i)$, *i.e.*, featuring a forked algebraic plan q_i that will compute the list contents.

Compilation Rule 17 for the pseudo SQO `Box()` establishes surrogates in column c, populates mapping ts accordingly, and thus implements the query plan forking explained in Section 3. Rule 18, defining the behavior of `UnBox()`, reverses the effect of `Box()` via a surrogate-based equi-join. We thus have $e\texttt{.Box().UnBox()} = e$, as expected.

Rule 7 compiles the comprehension $e_1\texttt{.Select}\,(v \texttt{=>} e_2)$ and thus forms the core of the LINQ compiler. Indeed, the compilation of other SQOs (*e.g.*, `Sum`, `SelectMany`, `Where`, `GroupBy`) is defined in terms of `Select`. Paragraph 2 (Appendix) discusses additional SQOs and their loop-lifted implementation. Here we point out the following:

$$\frac{\Gamma;\, loop \vdash e \Rightarrow (q, [c_1, c_2, \ldots, c_n, \ldots], ts)}{\Gamma;\, loop \vdash \texttt{e.n} \Rightarrow \left(\pi_{\mathsf{iter},\mathsf{pos},c_n}(q), [c_n], \begin{bmatrix}\{c_n \mapsto (q_n, cs_n, ts_n)\} & \text{if } c_n \mapsto (q_n, cs_n, ts_n) \in ts \\ \varnothing & \text{otherwise}\end{bmatrix}\right)} (1)$$

$$\frac{\Gamma;\, loop \vdash e_i \Rightarrow (q_i, cs_i, ts_i)\big|_{i=1,2} \quad q \equiv @_{\mathsf{pos}:1}\left(\pi_{\mathsf{iter},cs_1 \| cs_2}\left(q_1 \bowtie_{\mathsf{iter}} q_2\right)\right)}{\Gamma;\, loop \vdash \texttt{(}e_1\texttt{,}e_2\texttt{)} \Rightarrow (q, cs_1 \| cs_2, ts_1 \| ts_2)} (2)$$

$$\frac{\Gamma(v) = (q, cs, ts)}{\Gamma;\, loop \vdash v \Rightarrow (q, cs, ts)} (3) \qquad \frac{}{\Gamma;\, loop \vdash \begin{array}{l}\texttt{table}\ R\ \texttt{(}c_1\texttt{,}\ldots\texttt{,}c_n\texttt{)} \\ \texttt{with key (}c_{k_1}\texttt{,}\ldots\texttt{,}c_{k_m}\texttt{)}\end{array} \Rightarrow \left(loop \times \varrho_{\mathsf{pos}:\langle c_{k_1},\ldots,c_{k_m}\rangle}(\text{⛁}R), [c_1, \ldots, c_n], \varnothing\right)} (4)$$

$$\frac{\Gamma;\, loop \vdash e \Rightarrow (q, [c], \varnothing) \quad q_0 \equiv \pi_{\mathsf{iter},\mathsf{pos},c:c'}\left(\mathrm{CAST}_{c':(t)c}\,(q)\right)}{\Gamma;\, loop \vdash \texttt{(}t\texttt{)}e \Rightarrow (q_0, [c], \varnothing)} (5)$$

$$\frac{* \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{=}, \texttt{<}, \texttt{and}, \texttt{or}, \ldots\} \quad \Gamma;\, loop \vdash e_i \Rightarrow (q_i, [c_i], \varnothing)\big|_{i=1,2} \quad q \equiv \pi_{\mathsf{iter},\mathsf{pos},c:\mathsf{val}}\left(\circledast_{\mathsf{val}:\langle c_1,c_2\rangle}(q_1 \bowtie_{\mathsf{iter}} q_2)\right)}{\Gamma;\, loop \vdash e_1 * e_2 \Rightarrow (q, [c], \varnothing)} (6)$$

$$\frac{\begin{array}{c}\{\ldots, x \mapsto (q_x, cs_x, ts_x), \ldots\};\, loop \vdash e_1 \Rightarrow (q_1, cs_1, ts_1) \\ q_v \equiv \varrho_{\mathsf{inner}:\langle \mathsf{iter},\mathsf{pos}\rangle}(q_1) \quad map \equiv \pi_{\mathsf{iter},\mathsf{inner}}(q_v) \quad loop_v \equiv \pi_{\mathsf{iter}:\mathsf{inner}}(map) \\ \Gamma_{lift} \equiv \{\ldots, x \mapsto \left(\pi_{\mathsf{iter}:\mathsf{inner},\mathsf{pos},cs_x}(q_x \bowtie_{\mathsf{iter}} map), cs_x, ts_x\right), \ldots\} \\ \Gamma_{lift} + \{v \mapsto \left(@_{\mathsf{pos}:1}(\pi_{\mathsf{iter}:\mathsf{inner},cs_1}(q_v)), cs_1, ts_1\right)\};\, loop_v \vdash e_2 \Rightarrow (q_2, cs_2, ts_2)\end{array}}{\begin{array}{c}\{\ldots, x \mapsto (q_x, cs_x, ts_x), \ldots\};\, loop \vdash e_1\texttt{.Select(}v\ \texttt{=>}\ e_2\texttt{)} \Rightarrow \\ \left(\pi_{\mathsf{iter},\mathsf{pos},cs_2}(\varrho_{\mathsf{pos}:\langle \mathsf{inner}\rangle/\mathsf{iter}}(\pi_{\mathsf{inner}:\mathsf{iter},cs_2}(q_2) \bowtie_{\mathsf{inner}} map)), cs_2, ts_2\right)\end{array}} (7)$$

$$\frac{\Gamma;\, loop \vdash e_1\texttt{.Select(}v\ \texttt{=>}\ e_2\texttt{)} \Rightarrow (q_1, [c], \varnothing) \quad q_2 \equiv \mathrm{SUM}_{c:\langle c\rangle/\mathsf{iter}}(q_1) \quad q_0 \equiv @_{c:0}\left(loop \setminus \pi_{\mathsf{iter}}(q_2)\right)}{\Gamma;\, loop \vdash e_1\texttt{.Sum(}v\ \texttt{=>}\ e_2\texttt{)} \Rightarrow (@_{\mathsf{pos}:1}(q_0 \cup q_2), [c], \varnothing)} (8)$$

$$\frac{\Gamma;\, loop \vdash e_1\texttt{.Select(}v_1\ \texttt{=>}\ e_2\texttt{.Select(}v_3\ \texttt{=>}\ e_3\,[v_1/v_2]\texttt{))} \Rightarrow (q, cs, ts)}{\Gamma;\, loop \vdash e_1\texttt{.SelectMany(}v_1\ \texttt{=>}\ e_2\texttt{,(}v_2\texttt{,}v_3\texttt{)}\ \texttt{=>}\ e_3\texttt{)} \Rightarrow (q, cs, ts)} (9)$$

$$\frac{\Gamma;\, loop \vdash e_1 \Rightarrow (q_1, cs_1, ts_1) \quad \Gamma;\, loop \vdash e_1\texttt{.Select(}v\ \texttt{=>}\ e_2\texttt{)} \Rightarrow (q_2, [c], \varnothing) \quad q \equiv \pi_{\mathsf{iter},\mathsf{pos}:\mathsf{pos}_0,cs_1}\left(\varrho_{\mathsf{pos}_0:\langle \mathsf{pos}\rangle/\mathsf{iter}}\left(q_1 \bowtie_{\mathsf{iter},\mathsf{pos}} \pi_{\mathsf{iter},\mathsf{pos}}(\sigma_c(q_2))\right)\right)}{\Gamma;\, loop \vdash e_1\texttt{.Where(}v\ \texttt{=>}\ e_2\texttt{)} \Rightarrow (q, cs_1, ts_1)} (10)$$

$$\frac{\Gamma;\, loop \vdash e_1\texttt{.Select(}v_1\ \texttt{=>}\ e_2\texttt{)} \Rightarrow (q_1, cs_1, \varnothing) \quad \Gamma;\, loop \vdash e_1\texttt{.Select(}v_2\ \texttt{=>}\ e_3\texttt{)} \Rightarrow (q_2, cs_2, ts_2) \quad q_{\mathsf{g}} \equiv \pi_{\mathsf{iter},\mathsf{pos},cs_1,\mathsf{g}}\left(\varrho_{\mathsf{g}:\langle \mathsf{iter},cs_1\rangle}(q_1)\right) \quad q_i \equiv \pi_{\mathsf{iter}:\mathsf{g},\mathsf{pos}:\mathsf{pos}_0,cs_2}\left(\varrho_{\mathsf{pos}_0:\langle \mathsf{pos}\rangle/\mathsf{g}}(q_2 \bowtie_{\mathsf{iter},\mathsf{pos}} \pi_{\mathsf{iter},\mathsf{pos},\mathsf{g}}(q_{\mathsf{g}}))\right)}{\Gamma;\, loop \vdash e_1\texttt{.GroupBy(}v_1\ \texttt{=>}\ e_2\texttt{,}v_2\ \texttt{=>}\ e_3\texttt{)} \Rightarrow \left(\varrho_{\mathsf{pos}:\langle \mathsf{g}\rangle/\mathsf{iter}}\left(\delta(\pi_{\mathsf{iter},cs_1,\mathsf{g}}(q_{\mathsf{g}}))\right), [cs_1, \mathsf{g}], \{\mathsf{g} \mapsto (q_i, cs_2, ts_2)\}\right)} (11)$$

$$\frac{q_0 \equiv loop \times @_{\mathsf{pos}:1}\left(\pi_{\mathsf{hid}}\left(\sigma_{\mathsf{name}=name \wedge \mathsf{kind}=\mathtt{DOC}}(\text{⛁}\,\mathsf{XMLdocs})\right)\right)}{\Gamma;\, loop \vdash \texttt{XElement.Load(}name\texttt{)} \Rightarrow (q_0, [\mathsf{hid}], \varnothing)} (12) \qquad \frac{\Gamma;\, loop \vdash e \Rightarrow (q, [c], \varnothing) \quad q_0 \equiv \pi_{\mathsf{iter},\mathsf{pos},\mathsf{value}}\left(\text{⛁}\,\mathsf{XMLdocs} \bowtie_{\mathsf{hid}=c} q\right)}{\Gamma;\, loop \vdash e\texttt{.Value} \Rightarrow (q_0, [\mathsf{value}], \varnothing)} (13)$$

$$\frac{\Gamma;\, loop \vdash e \Rightarrow (q, [c], \varnothing)}{\Gamma;\, loop \vdash e\texttt{.Descendants(}name\texttt{)} \Rightarrow \left(\varrho_{\mathsf{pos}:\langle \mathsf{hid}\rangle/\mathsf{iter}}\left(\pi_{\mathsf{iter},\mathsf{hid}}\left((\sigma_{\mathsf{name}=name \wedge \mathsf{kind}=\mathtt{ELEM}}(\text{⛁}\,\mathsf{XMLdocs})) \bowtie_{\mathsf{hid}\,\mathrm{DescOf}\,c} q\right)\right), [\mathsf{hid}], \varnothing\right)} (14)$$

$$\frac{\Gamma;\, loop \vdash e \Rightarrow (q, [c], \varnothing)}{\Gamma;\, loop \vdash e\texttt{.Element(}name\texttt{)} \Rightarrow \left(\sigma_{\mathsf{pos}=1}\left(\varrho_{\mathsf{pos}:\langle \mathsf{hid}\rangle/\mathsf{iter}}\left(\pi_{\mathsf{iter},\mathsf{hid}}\left((\sigma_{\mathsf{name}=name \wedge \mathsf{kind}=\mathtt{ELEM}}(\text{⛁}\,\mathsf{XMLdocs})) \bowtie_{\mathsf{hid}\,\mathrm{ChildOf}\,c} q\right)\right)\right), [\mathsf{hid}], \varnothing\right)} (15)$$

$$\frac{\Gamma;\, loop \vdash e \Rightarrow (q, [c], \varnothing)}{\Gamma;\, loop \vdash e\texttt{.Attribute(}name\texttt{)} \Rightarrow \left(\sigma_{\mathsf{pos}=1}\left(\varrho_{\mathsf{pos}:\langle \mathsf{hid}\rangle/\mathsf{iter}}\left(\pi_{\mathsf{iter},\mathsf{hid}}\left((\sigma_{\mathsf{name}=name \wedge \mathsf{kind}=\mathtt{ATTR}}(\text{⛁}\,\mathsf{XMLdocs})) \bowtie_{\mathsf{hid}\,\mathrm{ChildOf}\,c} q\right)\right)\right), [\mathsf{hid}], \varnothing\right)} (16)$$

$$\frac{\Gamma;\, loop \vdash e \Rightarrow (q, cs, ts)}{\Gamma;\, loop \vdash e\texttt{.Box()} \Rightarrow \left(@_{\mathsf{pos}:1}(\pi_{\mathsf{iter},c:\mathsf{iter}}(loop)), [c], \{c \mapsto (q, cs, ts)\}\right)} (17) \qquad \frac{\Gamma;\, loop \vdash e \Rightarrow (q, [c], \{c \mapsto (q_c, cs_c, ts_c)\}) \quad q_0 \equiv \pi_{\mathsf{iter}:\mathsf{iter}_0,\mathsf{pos},cs_c}(\pi_{\mathsf{iter}_0:\mathsf{iter},\mathsf{iter}:c}(q) \bowtie_{\mathsf{iter}} q_c)}{\Gamma;\, loop \vdash e\texttt{.UnBox()} \Rightarrow (q_0, cs_c, ts_c)} (18)$$

Figure 7: Inference rules defining the algebraic compiler for basic LINQ expressions (Rules 1–6), SQOs (Rules 7–11), LINQ to XML methods (Rules 12–16), and (un)boxing (Rules 17–18). ‖ denotes list concatenation (see Rule 2).

- Rule 4 reads base table R and then derives a default ordering in column `pos` based on R's keys.
- Rules 14–16 use join predicates (*e.g.*, ChildOf) defined on XML node identifiers to implement the XPath-style navigation embodied by the LINQ to XML methods `Descendants`, `Element`, and `Attribute`. The rules use ϱ to derive list order (`pos`) from XML document order (`hid`) as required by the LINQ to XML semantics.

The algebraic plan bundle—comprised of q and the entries in ts—for LINQ program e is obtained through the evaluation of $\varnothing; \boxed{\begin{array}{c}\mathsf{iter}\\1\end{array}} \vdash e \Rightarrow (q, cs, ts)$. In the case of Program P, this leads to a trace of rule applications reviewed in Paragraph 3 (Appendix). In Paragraph 4 (Appendix) and [10, Section 4], we elaborate on the general loop-lifting technique—including the roles of Γ and the *loop* table.

With loop lifting, the complete family of LINQ SQOs comes into reach of database-supported execution. In the absence of a runtime-accessible encoding of order, the .NET LINQ providers either (1) compile order-sensitive SQOs into unordered substitute operations (`Concat` is translated into a SQL `UNION ALL` while `Take` turns into `SELECT TOP(·)` over an arbitrarily ordered table, for example), (2) resort to only partially implement the SQO semantics (*e.g.*, `Concat` and `Union` accept flat lists only), or (3) flag these SQOs as being unsupported. Table 2 summarizes the levels of SQO support in the Ferry-based as well as the .NET-supplied LINQ providers.

FERRY	Standard Query Operators	.NET
faithful	`Aggregate`, `All`, `Any`, `Average`, `Count`, `Contains`, `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`	partial
	`ElementAt[OrDefault]`, `Last[OrDefault]`, `Reverse`, `Select((`v`,`p`)=>...)`, `SelectMany((`v`,`p`)=>...)`, `SkipWhile`, `TakeWhile`, `Where((`v`,`p`)=>...))`, `Zip`	none

Table 2: Levels of database support for the SQO family in the FERRY-based and .NET-supplied LINQ providers.

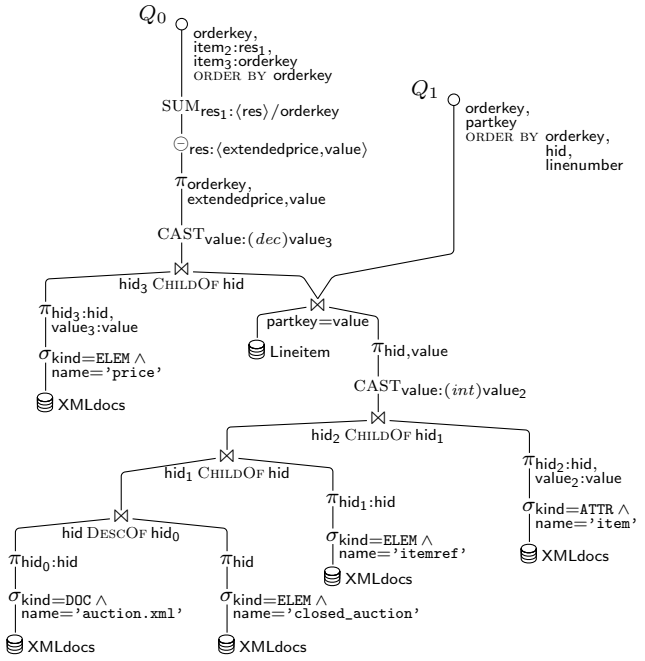


Figure 8: Optimized algebraic plan bundle $Q_{0,1}$ implementing Program P.

5. (PARTIAL) BUNDLE EXECUTION

Compiling Program P yields an initial algebraic query plan of about 150 operators. The plan features *two roots*, representing the outer list nesting level of the query result (Q_0) and the inner lists of `partkey`s (Q_1), respectively. At this point we are able to reuse proven query optimization and SQL:1999 code generation infrastructure: the FERRY-based LINQ provider shares its table algebra dialect with PATHFINDER [7, 10], a purely relational compiler for XQuery. PATHFINDER's optimizer is prepared to cope with query plans of the indicated size. Extensive data-flow analysis and operator property annotation—mainly based on properties like functional dependencies or column usage—is followed by a peephole-style plan simplification phase. In particular, maintenance of pos columns and associated ϱ operators are removed if the source program semantics does not depend on list order. The optimizer typically achieves a significant

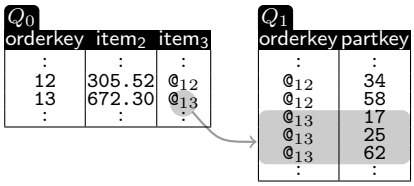


Figure 9: Tabular result bundle for Program P.

plan size reduction and aims to reshape the final plan to facilitate the subsequent SQL:1999 code generation [8].

Figure 8 depicts the optimized algebraic plan bundle (of < 30 operators), exhibiting the two plan roots (⚲) that result from query plan forking (Sections 3 and 4). The plan roots are fed into PATHFINDER's SQL:1999 code generator which applies a greedy strategy to derive a sequence of SQL statements—assembled to form a SQL common table expression (`WITH` $\cdots$)—that collectively implement the input plan. The final emitted SQL:1999 code for plans Q_0 and Q_1 is to be found in Paragraph 5 (Appendix). PATHFINDER has matured over the last seven years and has shown its capability to compile functional, LINQ-style languages [9]. Compiler, optimizer, and code generator are available in open source form [20].

The evaluation of the SQL code for Q_0 and Q_1 yields two tabular results as depicted in Figure 9. This table bundle holds Program P's result (see Paragraph 1 (Appendix)) according to the representation introduced in Section 3. The table schemata slightly deviate from the compiler's default iter|pos|item$_1$|$\cdots$: the optimizer succeeded in identifying columns other than the dedicated iter, pos to faithfully represent iteration and list order, respectively (see the ORDER BY annotations attached to the plan roots in Figure 8).

Parallelism and partial instantiation. Recall that query forking yields a bundle of *independent* queries. The back-end may thus execute the SQL queries of a bundle in any order it sees fit or, possibly, concurrently. This is in contrast to the avalanche of queries generated by the current .NET providers: in this case, the LINQ to XML provider iteratively invokes LINQ to SQL and thus effectively controls time and sequence of query submission. The LINQ to XML provider only resumes execution once SQL query evaluation completes.

Finally, query independence enables *partial object instantiation*. The contents of table Q_0 (Figure 9) on its own already permit the instantiation of C# objects of the form `new{order=`o`,gain=`g`,partkeys=`◯`}`. Here, ◯ represents a "hole", *i.e.*, a C# method to be executed once the `partkeys` field is actually accessed—only then would query Q_1 be submitted for execution by the database back-end.

6. IN THE LAB WITH THE BACK-ENDS

Loop lifting and avalanche safety promise a significant runtime impact that we try to quantify here. To this end, we married the well-known TPC-H [23] and XMark [19] benchmark data sets (Paragraph 1 (Appendix)) such that $\approx 48\,\%$ of all TPC-H line items find a matching auction item in the companion XMark document. We considered four different scale factors of these data sets, ranging from a total of 2 MB to 2 GB of data. The experiments were conducted under .NET Framework 4, hosted on a 2.66 GHz Intel Core 2 Duo computer with 3 GB of RAM (running Windows XP Profes-

Scale Factor (data set size)		.NET LINQ		FERRY-based LINQ		
			SQL Server		SQL Server	RDBMS X
XMark	TPC-H	# queries	⏱ (sec)	# queries	⏱ (sec)	⏱ (sec)
0.01 (1.1 MB)	0.001 (1.0 MB)	97	1.828	2	0.156	0.125
0.1 (11 MB)	0.01 (10 MB)	975	157.053	2	0.453	0.790
1 (110 MB)	0.1 (100 MB)	9 750	DNF	2	4.246	6.318
10 (1.1 GB)	1 (1.0 GB)	(97 500)	OOM	2	40.106	86.420

Table 3: Number of SQL queries emitted by the LINQ providers and observed wall-clock execution times, average of 10 runs (DNF: did not finish within hours, OOM: raises out of memory exception).

sional SP3). Microsoft SQL Server 2008 Enterprise was the primary database back-end.

We compiled Program P via the .NET-supplied as well as the FERRY-based LINQ providers and exposed SQL Server's Index Tuning Wizard to the resulting query workload. The proposed indexes were created, among these (1) a (partkey, orderkey) B-tree index on table Lineitem in-lining column extendedprice on the leaf level, and (2) a (name, kind) index on table XMLdocs in-lining column hid in the index leaves.

The observed wall-clock query execution times are summarized in Table 3. Nature and number of the SQL queries emitted by both providers have been monitored using the SQL Server Profiler. Almost identical behavior—in terms of query nature and number—is shown by LINQ to SQL and LINQ to Entities.

As anticipated, the query avalanche effect quickly overwhelms the relational back-end. With the .NET LINQ to XML provider in charge of the outer comprehension (line 2 in Figure 3), the LINQ to SQL provider is *iteratively* invoked to query table Lineitem (one invocation per `closed_auction` XML elements produced by the outer comprehension). Already for a medium scale factor and the associated 9 750 invocations, the query evaluation effort and expensive context switches between the involved LINQ providers add up to work that cannot be finished within hours (DNF in Table 3).

In contrast, the FERRY-based provider is unaffected by the asymmetry induced by the SQO call chain (recall Figure 1): with the entire LINQ program compiled into database-executable code, exactly two queries (Paragraph 5 (Appendix)) are shipped for evaluation by the back-end RDBMS . The number of queries issued remains constant with changing database instance size and can indeed be forecasted exactly by static type analysis—no query avalanche is set off.

Let us note that we consider issues of in-heap *vs.* database-supported XML processing secondary to the principal avalanche-safety goal.[2] Rules 12–16 (Figure 7) which compile LINQ to XML's methods, depend just as little on an ORDPATH-based node encoding as FERRY's SQL output is tied to run on SQL Server. Any SQL:1999-capable query engine using a faithful XML infoset representation and processor—be it native or relational—should make for an acceptable back-end. To make this point, we picked a different relational database system (coined "RDBMS X") off the shelf and used preorder ranking to encode the XMark XML instances (see Table 6; the predicates CHILDOF, DESCOF are easily adapted [10]). The rightmost column of Table 3 attests that we obtain a setup exhibiting avalanche safety along with the same beneficial performance characteristics.

[2]In fact, query avalanches may also hit programs that speak a *single* non-XML LINQ dialect only—the phenomenon is showcased in Paragraph 6 (Appendix).

7. RELATED WORK

The community's legitimate interest in the LINQ technology [2,21] is a continuation of the now decades-old search for a true amalgam of database queries and programming languages. With its comprehension-based, principled approach to iteration, LINQ lends itself to a formal approach to translation [3] and optimization. This marks an interesting spot on the long route that led from Pascal/R in the late 1970s, over relational approaches to program optimization [12], to today's language-integrated database abstractions like the ActiveRecord component of Ruby on Rails.

Microsoft further develops the LINQ idea towards LRX—LINQ over Relations and XML—in which a user-definable mapping from XML Schema to tables derives a tailored relational representation for a given XML instance [22]. In addition to table columns of atomic type, this representation will typically contain literal XML fragments, *i.e.*, columns of type `XML`, to (1) facilitate a full-fidelity re-serialization of the stored XML instance and (2) to allow the use of XPath or XQuery-specific operations that could not be evaluated over LRX's tabular XML representation alone (*e.g.*, `descendant` location steps). LRX translates incoming LINQ queries into a mix of SQL and embedded calls to Microsoft SQL Server's built-in XQuery engine.

In the light of LRX, the FERRY-based LINQ provider follows a somewhat more elementary approach: we define purely relational encodings of the LINQ data model and its operations (SQOs) and thus bring a wider range of LINQ constructs into the reach of the SQL query processor. This significantly reduces the number of context switches between SQL, XML, and in-heap processing phases. Further, the unified relational approach enables a whole-LINQ-program analysis and optimization that can reason over relational and XML-related program parts alike; this has been identified as an open future work item in [22].

The introduction of `Box()` and `UnBox()` by the compiler (Section 3) is inspired by techniques originally invented for the optimized representation of values of polymorphic type in programming languages [15]. Here, we let `Box()` control the forking of the compilation process that emits bundles of SQL queries. There is a correspondence with the operators ν/μ (or *nest*/*unnest*) introduced in the context of NF^2 databases [6,18]. However, operators ν/μ unfold their effect at *runtime* when a query is evaluated over a nested database, while `Box()`/`UnBox()` are *compile-time* concepts that guide code generation for a 1NF relational database system.

Our use of the surrogates $\mathbb{Q}_i$ resembles an ordered variant of van den Bussche's approach to the simulation of the nested algebra via the flat relational algebra [24]. While the simulation derives variable-width keys from the data itself, we employ compact ϱ-generated surrogates—the former approach would seamlessly plug into the present framework.

8. WRAP-UP

The present work describes the construction of an alternative LINQ query provider whose primary aim is to yield a faithful and efficient relational account of both, the static and the dynamic aspects of the LINQ data model. We saw that the LINQ semantics may be understood in terms of (1) tabular encodings of nested, ordered lists, records, as well as XML nodes, and (2) an algebraic compiler that can translate programs—built using the large family of LINQ standard query operators—into bundles of independent relational queries. This translation is true to the SQO semantics and does not forfeit operator characteristics (*e.g.*, the dependence on or the preservation of order).

The Ferry-based LINQ provider employs a compilation technique that embraces LINQ dialects over tabular as well as XML data. This uniformity overcomes provider asymmetry issues, one cause of iterative provider invocation (query avalanches), leading to an inefficient serial query-after-query mode of execution.

The algebraic LINQ compiler builds on a combination of loop lifting and non-parametric value representations to emit query bundles whose size is only dependent on the result type of the compiled program.

Work in flux. Beyond querying, a complete LINQ provider facilitates object updates. Support for the required update tracking in Ferry currently lies on the workbench.

As of today, Ferry's optimizer turns to the constituent queries of a bundle in isolation. Although this already goes a long way in removing obsolete plan fragments (order maintenance, in particular), a whole-bundle view of optimization clearly will further improve code quality. This relates to sharing as well as the reuse of existing table columns as surrogates ($@_i$).

The algebraic compiler of Figure 7 is not particularly tied to the specifics of LINQ. Its design is sufficiently generic to allow adaptations that support similar language-integrated query facilities. We develop a Ruby-based variant of Ferry that will significantly extend the limits of database support currently available in ActiveRecord and its successor ARel in the upcoming Ruby on Rails 3.0. We further inject Ferry into the functional Links [5] multi-tiered programming framework. This work will also serve as a playground to experiment with true support for higher-order functions [17] in Ferry.

9. REFERENCES

[1] A. Adya, J. Blakeley, S. Melnik, and S. Muralidhar. Anatomy of the ADO.NET Entity Framework. In *Proc. SIGMOD*, 2007.

[2] R. Agrawal, A. Ailamaki, P. Bernstein, E. Brewer, M. Carey, S. Chauduri, A. Doan, D. Florescu, M. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. Haas, A. Halevy, and J. Hellerstein. The Claremont Report on Database Research. *CACM*, 52(6), 2009.

[3] G. M. Bierman, E. Meijer, and M. Torgersen. Lost In Translation: Formalizing Proposed Extensions to C#. In *Proc. OOPSLA*, 2007.

[4] M. Chakravarty, R. Leshchinskiy, S. Peyton-Jones, G. Keller, and S. Marlow. Data-Parallel Haskell: A Status Report. In *Proc. DAMP*, 2007.

[5] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. In *Proc. FMCO*, 2006.

[6] P. Fischer and S. Thomas. Operators for Non-First-Normal-Form Relations. In *Proc. COMPSAC*, 1983.

[7] T. Grust, M. Mayr, and J. Rittinger. Let SQL Drive the XQuery Workhorse. In *Proc. EDBT*, 2010.

[8] T. Grust, M. Mayr, J. Rittinger, S. Sakr, and J. Teubner. A SQL:1999 Code Generator for the Pathfinder XQuery Compiler. In *Proc. SIGMOD*, 2007.

[9] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: Database-Supported Program Execution. In *Proc. SIGMOD*, 2009.

[10] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB*, 2004.

[11] S. P. Jones and P. Wadler. Comprehensive Comprehensions: Comprehensions with "Order by" and "Group by". In *Proc. Haskell Workshop*, 2007.

[12] D. Lieuwen and D. DeWitt. A Transformation-based Approach to Optimizing Loops in Database Programming Languages. In *Proc. SIGMOD*, 1992.

[13] E. Meijer. Confessions of a Used Programming Language Salesman: Getting the Masses Hooked on Haskell. *ACM SIGPLAN Notices*, 42(10), 2007.

[14] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Objects, Relations, and XML in the .NET Framework. In *Proc. SIGMOD*, 2006.

[15] A. Ohori. Type-Directed Specialization of Polymorphism. *Information and Computation*, 155(1-2), 1999.

[16] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. SIGMOD*, 2004.

[17] J. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation*, 11(4), 1998.

[18] H. J. Schek and M. H. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 11(2), 1986.

[19] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB*, 2002.

[20] T. Schreiber, S. Bonetti, T. Grust, M. Mayr, and J. Rittinger. Thirteen New Players in the Team: A Ferry-based LINQ to SQL Provider. In *Proc. VLDB*, 2010.

[21] M. Stonebraker, J. Becla, D. Dewitt, K. Lim, D. Maier, O. Ratzesberger, and S. Zdonik. Requirements for Science Data Bases and SciDB. In *Proc. CIDR*, 2009.

[22] J. F. Terwilliger, P. A. Bernstein, and S. Melnik. Full-Fidelity Flexible Object-Oriented XML Access. In *Proc. VLDB*, 2009.

[23] TPC Benchmark H. TP Council. `http://www.tpc.org/tpch/`.

[24] J. van den Bussche. Simulation of the Nested Relational Algebra by the Flat Relational Algebra, with an Application to the Complexity of Evaluating Powerset Algebra Expressions. *Theoretical Computer Science*, 254(1-2), 2001.

[25] P. Wadler. Comprehending Monads. In *Proc. LFP*, 1990.

```
1  from auction in XElement.Load("auction.xml")
2                          .Descendants("closed_auction")
3  from li      in db.GetTable<Lineitem>()
4  where (int)auction.Element("itemref").Attribute("item") ==
5        li.partkey
6  group new { partkey = li.partkey,
7              diff    = li.extendedprice -
8                        (decimal)auction.Element("price") }
9    by li.orderkey into g
10 select new { order    = g.Key,
11              gain     = g.Sum(t => t.diff),
12              partkeys = g.Select(t => t.partkey) }
```

Figure 10: Program P expressed in the user-facing LINQ query syntax. The C# compiler converts this form into an expression tree of chained SQO invocations (shown in Figure 1).

Lineitem

orderkey	partkey	extendedprice	shipmode
:	:	:	:
12	34	240.73	RAIL
12	58	187.00	SHIP
13	17	312.42	TRUCK
13	25	389.99	RAIL
13	62	246.50	SHIP
:	:	:	:

Table 4: Sample of a TPC-H Lineitem table [23].

```
<closed_auctions>
  <closed_auction>
    <itemref item="34"/>
    <price>122.21</price>
        ⋮
  </closed_auction>
        ⋮
</closed_auctions>
```

Figure 11: Excerpt of XMark [19] XML instance `auction.xml`.

XMLdocs

hid	kind	name	value
:	:	:	:
/1/1/6	ELEM	closed_auctions	"122.21..."
/1/1/6/1	ELEM	closed_auction	"122.21..."
/1/1/6/1/1	ELEM	itemref	""
/1/1/6/1/1/1	ATTR	item	"34"
/1/1/6/1/2	ELEM	price	"122.21"
/1/1/6/1/2/1	TEXT	—	"122.21"
:	:	:	:

Table 5: XML infoset encoding: ORDPATH identifiers [16] in column hid.

XMLdocs

pre	size	level	ki
:	:	:	:
812	96	2	EL
813	31	3	EL
814	1	4	EL
815	0	5	ATT
816	1	4	ELE
817	0	5	TEX
:	:	:	:

Table 6: XML infoset encoding: preorder ranks [10].

$$\frac{\Gamma;\, loop \vdash e \Rrightarrow (q, cs, ts) \qquad q_0 \equiv \pi_{\mathsf{iter},\mathsf{pos}:\mathsf{pos}_0,cs}\left(\varrho_{\mathsf{pos}_0:\langle -\mathsf{pos}\rangle/\mathsf{iter}}(q)\right)}{\Gamma;\, loop \vdash e\texttt{.Reverse()} \Rrightarrow (q_0, cs, ts)}\ (19)$$

$$\frac{\Gamma;\, loop \vdash e_1 \Rrightarrow (q_2, cs_2, ts_2) \quad \Gamma;\, loop \vdash e_2 \Rrightarrow (q_1, [c], \varnothing) \qquad q_0 \equiv \pi_{\mathsf{iter},\mathsf{pos},cs_2}\left(\sigma_{\mathsf{pos}\leqslant c}(q_1 \bowtie_{\mathsf{iter}} \pi_{\mathsf{iter},c}(q_2))\right)}{\Gamma;\, loop \vdash e_1\texttt{.Take(}e_2\texttt{)} \Rrightarrow (q_0, cs, ts)}\ (20)$$

$$\frac{\begin{array}{c}\{\ldots, x \mapsto (q_x, cs_x, ts_x), \ldots\};\, loop \vdash e_i \Rrightarrow (q_i, cs_i, ts_i)\,\big|_{i=1,2} \\ q \equiv \pi_{\mathsf{iter},\mathsf{pos},cs_1 \| cs_2}\left(q_1 \bowtie_{\mathsf{iter},\mathsf{pos}} q_2\right) \\ q_v \equiv \varrho_{\mathsf{inner}:\langle \mathsf{iter},\mathsf{pos}\rangle}(q) \quad map \equiv \pi_{\mathsf{iter},\mathsf{inner}}(q_v) \quad loop_v \equiv \pi_{\mathsf{iter}:\mathsf{inner}}(map) \\ q_{v_1} \equiv @_{\mathsf{pos}:1}(\pi_{\mathsf{iter}:\mathsf{inner},cs_1}(q_v)) \quad q_{v_2} \equiv @_{\mathsf{pos}:1}(\pi_{\mathsf{iter}:\mathsf{inner},cs_2}(q_v)) \\ \Gamma_{lift} \equiv \{\ldots, x \mapsto \left(\pi_{\mathsf{iter}:\mathsf{inner},\mathsf{pos},cs_x}(q_x \bowtie_{\mathsf{iter}} map), cs_x, ts_x\right), \ldots\} \\ \Gamma_{lift} + \{v_1 \mapsto (q_{v_1}, cs_1, ts_1), v_2 \mapsto (q_{v_2}, cs_2, ts_2)\};\, loop_v \vdash e_3 \Rrightarrow (q_3, cs_3, ts_3)\end{array}}{\begin{array}{c}\{\ldots, x \mapsto (q_x, cs_x, ts_x), \ldots\};\, loop \vdash e_1\texttt{.Zip(}e_2\texttt{,(}v_1\texttt{,}v_2\texttt{) => }e_3\texttt{)} \Rrightarrow \\ \left(\pi_{\mathsf{iter},\mathsf{pos},cs_3}(\varrho_{\mathsf{pos}:\langle \mathsf{inner}\rangle/\mathsf{iter}}(\pi_{\mathsf{inner}:\mathsf{iter},cs_3}(q_3) \bowtie_{\mathsf{inner}} map)), cs_3, ts_3\right)\end{array}}\ (21)$$

Figure 12: Extension of the algebraic LINQ compiler of Figure 7: implementations of the order-sensitive SQOs `Reverse`, `Take`, and `Zip`.

APPENDIX

1 A marriage of TPC-H and XMark. What do you gain if you bid for selected items on an Internet auction site instead of ordering them from your usual supplier? The sample LINQ Program P of Figure 10 tries to answer this question by joining TPC-H table Lineitem with XMark auction data (samples shown in Table 4 and Figure 11). To facilitate this marriage, we made sure that selected Lineitem.partkeys and XMark `@item` attributes can match: in the XMark data, we removed the `"item"` prefix in `@item` attribute values (*e.g.*, `"item34"` → `"34"`) to facilitate the comparison with TPC-H's integer partkey*s*. While TPC-H tables live in a relational database by default, we further populated the database with table XMLdocs, containing the relational XML infoset encoding of XMark instances (Figure 11). Tables 5 and 6 show equivalent samples of two such encodings, one based on ORDPATH-style hierarchical identifiers [16] as provided by SQL Server, one based on preorder traversal ranks [10]. Any other faithful XML infoset encoding would do, however (see Section 6).

The program of Figure 10 has been formulated in the user-facing LINQ *query syntax* which, most notably, features the `from-in-where-select` comprehension construct. Before the LINQ provider takes over, the C# compiler turns the query syntax into *expression trees* of chained invocations of Standard Query Operators (also referred to as *method syntax*, shown in Figure 1). Based on the table data of Tables 4 and 5, Program P will compute a result of type $[(\texttt{int}, \texttt{decimal}, [\texttt{int}])]$:

```
[... (12, 305.52, [34, 58]),
     (13, 672.30, [17, 52, 62]), ...]
```

(read: for order `12` with parts `34` and `58`, bidding on the auction site will yield a gain of `305.52`).

2 Faithful loop-lifted implementations of LINQ's `Reverse`, `Take`, and `Zip` SQOs. The inference Rules 19–21 of Figure 12 plug into the algebraic LINQ compiler introduced in Section 4 and add faithful support for the order-sensitive SQOs `Reverse`, `Take`, and `Zip`. Analogous compilation rules for `ElementAt`, `First`, and `Skip` follow immediately from Rule 20 (`Take`) if the positional predicate $\mathsf{pos} \leqslant c$ is adapted accordingly.

The `Zip` SQO has been added with the C# 4.0 release: $e_1\texttt{.Zip(}e_2\texttt{, (}v_1\texttt{,}v_2\texttt{) => }e_3\texttt{)}$ moves and applies the anonymous "slider" function $\texttt{(}v_1\texttt{,}v_2\texttt{) => }e_3$ over its input lists and effectively performs a *positional join* of $e_{1,2}$. This is directly reflected by the join $\bowtie_{\mathsf{iter},\mathsf{pos}}$ in Rule 21.

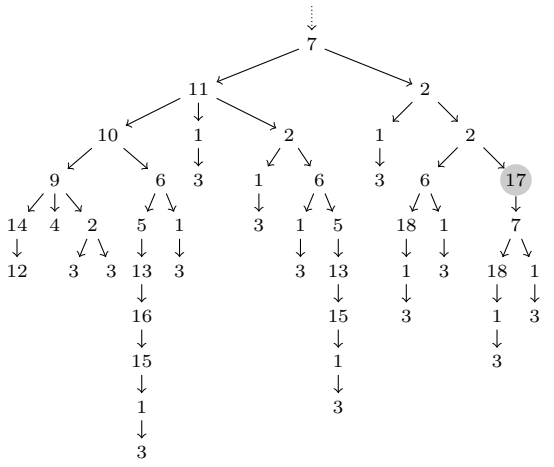


Figure 13: Inference rule applications traced while compiling the LINQ program of Figure 3 (read from top to bottom: Rule 7 is applied first). 17 marks the application of the `Box()` rule.

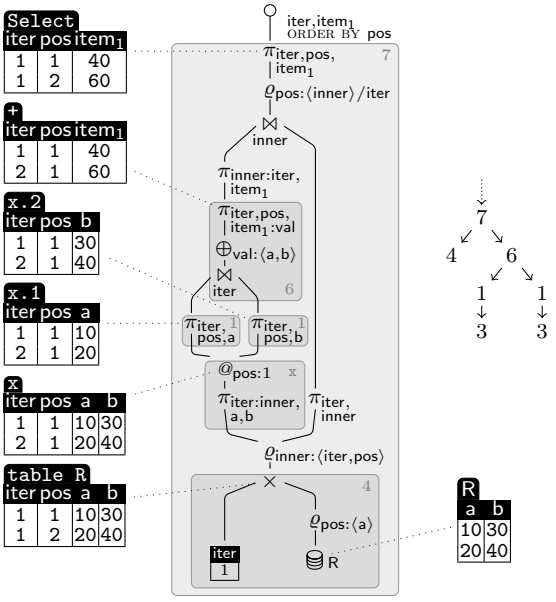


Figure 14: Unoptimized query plan and inference rule application trace for the simple query `(table R(a,b) with key(a)).Select(x => x.1 + x.2)`. Table annotations indicate intermediate results for subexpressions of the query.

3 Rule trace. Figure 13 traces all rule invocations performed while compiling Program P. The trace nodes refer to the compilation rules of Figure 7. The compiler initially invokes Rule 7 of Figure 7 to translate the outermost `Select` and then recursively applies rules to compile subexpressions. The algebraic plan is synthesized in a bottom-up fashion. When `Box()` is compiled via Rule 17, the compiler forks the compilation process once, leading to a plan bundle of two algebraic queries ($Q_{0,1}$ in Figure 8).

4 Loop-Lifted Compilation. We use the simple example expression

```
(table R(a,b) with key(a)).Select(x => x.1 + x.2) ,
```

its expression tree and the annotated inference rule trace in Figure 14 to provide an intuition of loop-lifting.

Starting with an empty variable environment Γ and a singleton table $loop \equiv$ iter 1, the compiler invokes Rule 7 (refer to the compilation rules of Figure 7). The antecedent of Rule 7 invokes Rule 4 to initiate the compilation of the `Select` SQO's binding expression `table R(a,b) with key(a)`. This binding expression is of type $[(int, int)]$. Maintaining the central compiler invariant (see Section 4), Rule 4 emits a piece of algebraic plan code that evaluates to a table with schema iter|pos|a|b of $2 + 2$ columns in which columns iter and pos contain information about iteration and list order (see Section 3), and columns a and b carry the actual contents of the rows in `R` (see `table R` in Figure 14). Observe how (1) iteration order is determined by the current *loop* relation, and (2) list order is derived from the keys of table `R`. The binding subexpresssion `table R(a,b) with key(a)` is to be evaluated *once* only. The second row $\langle 1, 2, 20, 40\rangle$ in `table R` thus is to be interpreted as "in the first (and only) evaluation of this expression, the value at the second list position is `(20,40)`".

From `table R`, Rule 7 derives a sequence of (two) bindings for the `Select` SQO's bound variable `x` (see query plan section `x` and the resulting `table x` in Figure 14). The `Select`'s body expression `x.1 + x.2` is evaluated *two* times, once for each such binding of variable `x` (column iter $\in \{1, 2\}$ in `table x`). `table x` serves as a representation of the bindings of `x` and is inserted into variable environment Γ by Rule 7. This environment is later consulted by Rule 3 to resolve variable references.

The body expression `x.1 + x.2`, an application of the binary operator `+`, is compiled under the control of Rule 6. The antecedent of this rule recursively compiles the two arguments, both of which are positional field accesses (handled by Rule 1; see the two plan sections marked 1 in Figure 14). Before the actual arithmetics via operator $\oplus$ is performed in plan section 6, an equi-join on column iter merges the two tables `x.1` and `x.2` that carry the arguments of the addition. Note how the resulting table `+` represents the result of the addition for *all evaluations* of the `Select` body: in the second iteration (iter $= 2$), the result is the *int* value 60 ($=$ `20 + 40`), for example.

Finally, Rule 7 transforms table `+` into table `Select` in Figure 14. While table `+` encodes the result of the `Select` SQO's *body* expression, table `Select` represents the result of the `Select` SQO itself: the result is a single list (iter $= 1$) of length two (pos $\in \{1, 2\}$).

5 Final SQL code generator output for Program P. Figures 15 and 16 show the SQL:1999 code that has been generated from the algebraic query pair $Q_{0,1}$ of Figure 8. Like $Q_{0,1}$ themselves, the SQL queries are independent of each other and permit concurrent execution by the back-end RDBMS. Executing the code of Figure 15 (*i.e.*, Q_0) only admits the partial instantiation of the resulting C# objects (see Section 5).

6 Query avalanches in single-LINQ-dialect programs. The query avalanche issue may hit programs that exclusively rely on the .NET LINQ to SQL provider. To see this, consider the LINQ program of Figure 17 that groups line items by mode of shipment to prepare delivery. No XML processing is involved: table Lineitem is the only (tabular) data source and the program is executed under the control of the LINQ

```
SELECT doc1.orderkey AS item1,
       SUM (doc1.extendedprice -
            CAST (doc6.value as DECIMAL)) AS item2,
       doc1.orderkey AS item3
  FROM Lineitem AS doc1,
       XMLdocs  AS doc2, XMLdocs AS doc3,
       XMLdocs  AS doc4, XMLdocs AS doc5,
       XMLdocs  AS doc6
 WHERE doc2.kind = DOC
   AND doc2.name = 'auction.xml'
   AND doc3.kind = ELEM
   AND doc3.name = 'closed_auction'
   AND doc3.hid.IsDescendantOf(doc2.hid) = 1
   AND doc4.kind = ELEM
   AND doc4.name = 'itemref'
   AND doc4.hid.GetAncestor(1) = doc3.hid
   AND doc5.kind = ATTR
   AND doc5.name = 'item'
   AND doc5.hid.GetAncestor(1) = doc4.hid
   AND doc1.partkey = CAST(doc5.value AS INTEGER)
   AND doc6.kind = ELEM
   AND doc6.name = 'price'
   AND doc6.hid.GetAncestor(1) = doc3.hid
 GROUP BY doc1.orderkey
 ORDER BY doc1.orderkey ASC;
```

Figure 15: SQL code generated for query Q_0 (Figure 8).

```
SELECT doc1.orderkey AS iter,
       doc1.partkey AS item1
  FROM Lineitem AS doc1,
       XMLdocs  AS doc2, XMLdocs AS doc3,
       XMLdocs  AS doc4, XMLdocs AS doc5
 WHERE doc2.kind = DOC
   AND doc2.name = 'auction.xml'
   AND doc3.kind = ELEM
   AND doc3.name = 'closed_auction'
   AND doc3.hid.IsDescendantOf(doc2.hid) = 1
   AND doc4.kind = ELEM
   AND doc4.name = 'itemref'
   AND doc4.hid.GetAncestor(1) = doc3.hid
   AND doc5.kind = ATTR
   AND doc5.name = 'item'
   AND doc5.hid.GetAncestor(1) = doc4.hid
   AND doc1.partkey = CAST(doc5.value AS INTEGER)
 ORDER BY doc1.orderkey ASC, doc3.hid ASC,
          doc1.linenumber ASC;
```

Figure 16: SQL code generated for query Q_1 (Figure 8).

to SQL provider.

The provider applies a staged and iterative evaluation strategy (see Algorithm 1). For n distinct orders, the LINQ to SQL provider thus submits between $1+2\times n$ and $1+4\times n$ SQL queries overall.[3] The exact number is only determined at runtime, dependent on the size and value distribution of the queried TPC-H instance. Note that the SQL queries in the resulting batch exhibit dependencies: the query of line 1 and the iterated queries of lines 3, 5 yield parameter marker bindings required to run subsequent queries. In contrast, the query batch is executed sequentially. Table 7 documents the immense performance impact of these data dependencies. The figures in this table were obtained using the experimental setup described in Section 6.

```
from li in db.GetTable<Lineitem>()
group li by li.orderkey into order
let shipment = from o in order
               group new { orderkey = o.orderkey,
                           partkey  = o.partkey  }
                  by o.shipmode
select new {
 order  = order.Key,
 byrail = shipment.FirstOrDefault(s => s.Key == "RAIL"),
 byship = shipment.FirstOrDefault(s => s.Key == "SHIP") }
```

Figure 17: LINQ program: the line items of each order are grouped by mode of shipment (by rail/by ship).

[3]The query avalanche phenomenon is an instance of the infamous, yet appropriately named *1+n Query Problem* that affects many implementations of object-relational mappers.

```
1 Query table Lineitem for the set of distinct orders
  (assume that this yields n orderkeys).          ⟨1 query⟩
2 For each of these n orders,
3 |  if the set of line items to be shipped by RAIL is
  |  non-empty then                                ⟨n queries⟩
4 |  |  retrieve those line items,                 ⟨0 ... n queries⟩
5 |  if the set of line items to be shipped by SHIP is
  |  non-empty then                                ⟨n queries⟩
6 |  |  retrieve those line items.                 ⟨0 ... n queries⟩
```

Algorithm 1: Iteration in the .NET LINQ to SQL provier.

With the Ferry-based provider, the query batch size is determined at compile-time, reflecting the number of list type constructors in the program's result *type*. Here, this type reads $[(\texttt{int},(\texttt{int},[(\texttt{int},\texttt{int})]),(\texttt{int},[(\texttt{int},\texttt{int})]))]$—or $[[\cdot][\cdot]]$ if we consider list type constructors only—invariably yielding a query bundle of size 3 (Table 7). The queries in this bundle are independent, allowing for the on-demand or partial instantiation of the query result (Section 5): a C# program that only consumes the `order` and `byrail` record components, effectively performing a projection on type $[[\cdot]]$, will execute only 2 of the 3 bundle queries. The current .NET LINQ to SQL provider still submits the whole batch in this case.

[To wrap this discussion up: for the program of Figure 17, LINQ to Entities crafts a single query of considerable complexity (246 lines of SQL code *vs.* 8+22+22 lines in Ferry's bundle of SQL queries). Nested lists are represented "inline" line", at the price of a wide result schema, data redundancy with respect to the outer nesting level, and ubiquitous `NULL` values. (It appears that a variant of this representation could be obtained from Ferry's three-queries bundle by a two-way outer-join.) Being monolithic, the LINQ to Entities-generated SQL query precludes partial instantiation. For a TPC-H instance of factor 0.1 (100 MB), the query requires 14.124 s to compute the result of the program of Figure 17.]

Scale Factor (# orders) TPC-H		.NET LINQ to SQL SQL Server # queries	(sec)	Ferry-based LINQ SQL Server # queries	(sec)
0.001	(1 500)	4 258	4.012	3	0.327
0.01	(15 000)	43 030	364.800	3	0.422
0.1	(150 000)	430 496	> 5 hrs	3	3.100
1	(1 500 000)	4 303 502	DNF	3	55.589

Table 7: Number of SQL queries emitted and observed wall-clock execution times for the LINQ program of Figure 17 (DNF: did not finish within 24 hours).