Ferry — Database-Supported Program Execution

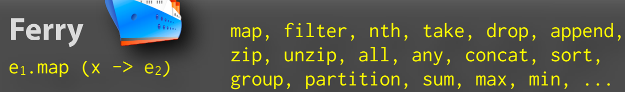
ID	NAME	DEPT	SALARY
1	Alex	DE	300
2	Bert	DE	100
3	Cora	DE	200
4	Drew	US	200
5	Erik	US	400
6	Fred	US	300
7	Gina	US	200
8	Herb	NL	600
9	Ivan	NL	400
10	Jill	UK	500

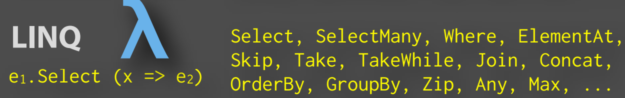
Your Programming Language's New Friend

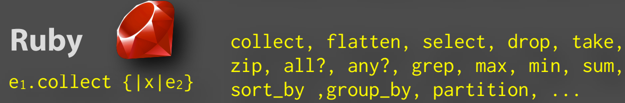
Ferry explores query compilation technology that can turn off-the-shelf relational database systems into efficient and scalable co-processors for your programming language's runtime.

(1) The **host PL prescribes data types and operations**,

(2) **Ferry compiles fragments of data–intensive computation into bundles of SQL:1999 queries**—for efficient, possibly parallel, execution on an associated database back-end.







With Ferry, you continue to use the idioms and operations of your favorite programming language to describe computation over **ordered** and **arbitrarily nested** records, lists, arrays, or maps.

Ferry's **list comprehension-based** compilation scheme is as compositional as your programming language's semantics and will not break when facing complex programs.

Order and Nesting on 1NF RDBMSs

Ferry has been designed to support the compilation and execution of programs over ordered, nested data structures. To realize this on top of unordered First Normal Form (1NF) SQL back-ends,

(1) Ferry **embeds order information in the data** itself (if relevant), and

(2) represents **nested data structures via flat tables containing surrogate key values**—much like in the 1980s.

Count the [·]s to Count the Queries

The (static) **type of your program's result** determines **how many database queries Ferry will emit** to compute that result. Nothing else impacts the query bundle size. Period.

If the result type *T* features *n* list constructors [·], Ferry prepares a bundle of *n* **independent queries** for execution.

(With Microsoft's LINQ to SQL, the number of submitted queries may depend on the database instance size. Yes, that may be *many* queries.)

Example: Result Type *T* ($n = 4$)

`[(int,[str],[(bool,[(int,int)])])]`

Programming with Tables

"Who are the two top-paid staff members per department in Table Employees?"

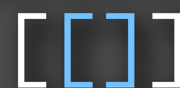
Ferry permits a compositional programming style that glues operations over tables and ordered lists, *e.g.*, `zip(·,·)`. Nesting occurs naturally. Somewhat close to Haskell. And XQuery. And LINQ. And SQL.

Ferry Sample Program *P*: Two Top-Paid Employees per Department

```
let e = table Employees (id int, name string,
                         dept string, salary int)
        with key (id)
in
  for x in e
  group by x.dept
  return (the (x.dept),
          take (2, for y in zip (x.name, x.salary)
                   order by y.salary descending
                   return y))
```

Result Type of Program *P*

`[(str,[str,int])]`



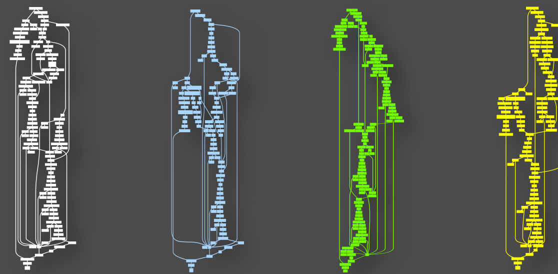
Intermediate Language: A Table Algebra Dialect

Operator	Meaning
⚲	serialization point (sub-plan root)
$\pi_{a_1:b_1,\ldots,a_n:b_n}$	project onto columns b_i, rename b_i into a_i
σ_p	select rows satisfying predicate p
$@_{a:v}$	attach column a containing constant value v
$_ \times _$, $_ \bowtie_p _$	Cartesian product, join with predicate p
$_ \dot\cup _$, $_ \setminus _$	disjoint union, difference
δ	eliminate duplicate rows
$\varrho_{a:\langle b_1,\ldots,b_n\rangle/c}$	group rows by c, then attach row rank in a
$\circledast_{a:\langle b_1,\ldots,b_n\rangle}$	attach result of application $*(b_1,\ldots,b_n)$ in a
$agg_{a:\langle b\rangle/c}$	group rows by c, then attach aggregated b in a
⛁$_R$	access database-resident table R
a b c literal table	literal table with columns a,b,c

Ferry compiles programs into a particularly simple dialect of a table algebra whose design **mimics the actual capabilities of modern relational query processors**. (No wishful thinking here.)

The compiler implements **loop lifting**, a translation technique that derives **set-oriented plans from iterative programs**. Primary goal: operate the database back-end in its efficient bulk processing mode.

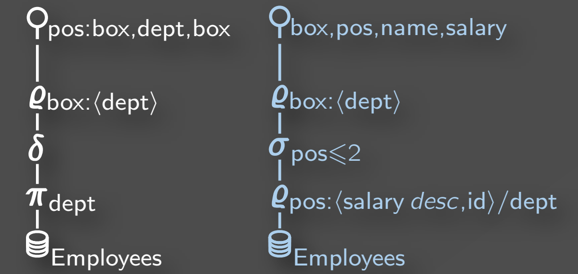
Shape of Algebraic Plan Bundle for Type *T*



Ferry's algebraic compiler yields a bundle of **independent** queries. The database back-end **may execute these queries in any order** it sees fit—or even in parallel.

(With Microsoft's LINQ to SQL, your back-end may experience an avalanche of sequentially submitted queries, each entailing an execution context switch.)

Algebraic Plan Bundle for Program *P*



Two independent algebraic plans compute the list contents for their associated list constructors. (Note: Plan ■ computes a single list, Plan ■ computes the contents of *many* nested lists—with **column box**, we may stitch the pieces together later on.)

In **column pos**, the plans embed **order in the data**: positional access (`take(n,·)`) or alignment (`zip(·,·)`) indeed makes sense.

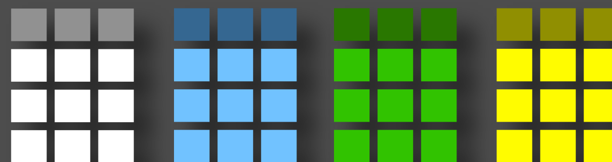
Target Language: SQL:1999, Off the Shelf

SQL	Purpose
`[ROW_NUMBER()|DENSE_RANK()] OVER (PARTITION BY ... ORDER BY ...)`	implementation of order semantics and nested lists
`[RIGHT OUTER] JOIN`	implementation of iteration
`UNION ALL`	sequence construction
`WITH ... AS ...`	basic block forming

Ferry comes equipped with a **code generator deriving standard SQL:1999 common table expressions** from algebraic plans.

Such code generators have been devised for IBM DB2 V9, MS SQL Server 2008, CWI's MonetDB column store, and kx Systems' kdb+. If your query engine can order and then number table rows, it is most probably a suitable Ferry back-end.

Lazy and/or Partial Plan Evaluation



The result has a **sensible interpretation per table**, each representing **all lists** of its associated type constructor. Table ■ already delivers

`[(42,@1,@2), (43,@3,@4), (44,@5,@6), …]`

If your program does not inspect lists beyond nesting level 1 yet, there is no need to compute Tables ■, ■, or ■ right now.

Fully Evaluated Result of Program *P*

pos	dept	box
1	DE	$@_1$
2	US	$@_2$
3	NL	$@_3$
4	UK	$@_4$

box	pos	name	salary
$@_1$	1	Alex	300
$@_1$	2	Cora	200
$@_2$	1	Erik	400
$@_2$	2	Fred	300
$@_3$	1	Herb	600
$@_3$	2	Ivan	400
$@_4$	1	Jill	500

Both **tables contribute pieces to the result** of Program *P* and may be sensibly computed, consumed, and interpreted on their own.

The **surrogate keys $@_k$ in columns box** enable the runtime system to assemble the complete result. And: without columns **box|pos**, Ferry's encoding of **flat unordered lists and regular DBMS tables coincide**.

Ferry vs. LINQ to SQL vs. Ruby— How Much Faster Do You Want To Be Today?

Queries Emitted for Program *P*

# Depts.	Ferry # Queries	LINQ # Queries
1 000	2	1 001
10 000	2	10 001
100 000	2	100 001

Execution Times for Program *P*

# Depts.	Ferry (sec)	LINQ (sec)
1 000	0.140	0.265
10 000	0.296	1.230
100 000	1.948	10.486

Ruby 1.9 Variant of Program *P*

# Depts.	Ruby (Heap Execution) (sec) load + execute
1 000	0.188 + 0.031
10 000	1.240 + 0.350
100 000	11.600 + 8.312

Ferry supports a data model and language semantics that is so close to Microsoft's **Language Integrated Query (LINQ)** that you can use Ferry as a replacement for the Microsoft LINQ to SQL provider. If you do, the **family of applicable Standard Query Operators (SQOs) grows**, all SQOs adhere to **proper order semantics**, and **no query avalanches** will be generated.

If your programs process significant amounts of database-resident data, Database-Supported Program Execution can save the host PL's runtime system from a lot of work. And from a job it has not been designed for.

Ferry—Database-Supported Program Execution.

Heap Image of the Result of Program *P*

```
[
 ("DE", [("Alex",300),("Cora",200)]),
 ("US", [("Erik",400),("Fred",300)]),
 ("NL", [("Herb",600),("Ivan",400)]),
 ("UK", [("Jill",500)])
]
```

A merge of the sorted Tables ■ and ■, performing a left outer join, suffices to materialize the result on the heap of the host PL.

On-demand materialization may be implemented in terms of *holes*: just scan Table ■ to produce `[("DE",□), …]`, where hole □ represents a suspension that will scan Table ■ once it is referenced.

www.ferry-lang.org

Eberhard Karls Universität Tübingen

Torsten Grust torsten.grust@uni-tuebingen.de
Manuel Mayr manuel.mayr@uni-tuebingen.de
Jan Rittinger jan.rittinger@uni-tuebingen.de
Tom Schreiber tom.schreiber@uni-tuebingen.de