PRQL

Pipelined Relational Query Language

Aljaž Mur Eržen

Compiler developer @EdgeDB

```
from animals
filter height_cm > 20
take 10
join k = keepers (==keeper_id)
filter k.first_name == "Thomas"
sort animals.age
select {
    species = animals.species,
    keeper = k.name,
    f"I'm {species}, {animals.age} years old",
}
```

Overview

Flaws of SQL

Language for relations

Compiling queries

PRQL, the project

A deep dive into

Flaws of SQL

Origins

1970, Edgar F. Code: Relational model

→ Abstraction over data storage

→ Tuple relational calculus

Origins

Relation $\sim$ a set of tuples

$$\pi_{\text{track\_id,name,title}}(R) \qquad \sigma_{\text{track\_id}=5}(R)$$

$$R * S$$

Origins

1974, Donald D. Chamberlin & Raymond F. Boyce

→ SEQUEL

→ Not a “proper” programming language

Human-friendly syntax

```
SELECT
  DISTINCT name
FROM
  invoices
```

Human-friendly syntax

```
SELECT
  SUM(total)
FROM
  invoices
```

Human-friendly syntax

```
SELECT
  total / SUM(total) OVER () AS normalized_total
FROM
  invoices
```

Name resolution

```
SELECT title AS title_alias
FROM albums
```

Name resolution

```
SELECT title AS title_alias
FROM albums
WHERE title LIKE 'Do I Wanna %'
GROUP BY title_alias
ORDER BY title_alias
```

Name resolution

More rules:

– ORDER BY positionals

– Correlated subqueries

– LATERAL

Not really composable

```
SELECT album_id, COUNT(*)
FROM tracks
GROUP BY album_id
```

Not really composable

```
SELECT i.*, a.artist_id
FROM (
  SELECT album_id, COUNT(*)
  FROM tracks
  GROUP BY album_id
) AS i
JOIN albums a USING (album_id)
```

Relations and scalars

```
SELECT * FROM emp
```

```
SELECT min(emp_id) FROM emp
```

Relations and scalars

```
SELECT * FROM emp
WHERE emp_id = (
    SELECT min(emp_id) FROM emp
)
```

Relations and scalars

Depends on the database contents:

```
SELECT emp_id FROM emp
WHERE role = 'manager'
```

Relations and scalars

```
SELECT * FROM emp
WHERE emp_id = (
    SELECT emp_id FROM emp
    WHERE role = 'manager'
)
```

We Can Do Better Than SQL by EdgeDB

Against SQL by scattered-thoughts.net

LINQ – a pipelined language for .NET

FunSQL.jl – compositional construction of queries for Julia

Malloy – a language for describing data

Ecto – an ORM library for Elixir

Design of a new

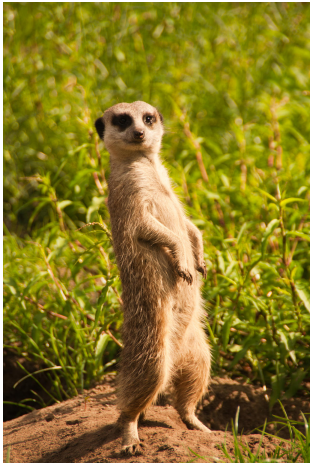
Language for relations

```
from animals
```

```
[
  {
    animal_id = 1,
    species = "Mus musculus",
    height_cm = 5,
    age = 1,
    keeper_id = 2
  },
  ... and 100 others ...
]
```

```
from animals
filter height_cm > 10
```

```
[
  {
    animal_id = 1,
    species = "Suricata suricatta",
    height_cm = 15,
    age = 3,
    keeper_id = 4
  },
  ... and 14 others ...
]
```



```
from animals
filter height_cm > 10
take 10
```

```
from animals
filter height_cm > 10
take 10
join keepers (
    animals.keeper_id == keepers.keeper_id
)
```

```
from animals
filter height_cm > 10
take 10
join k = keepers (
    animals.keeper_id == k.keeper_id
)
```

```
from animals
filter height_cm > 10
take 10
join k = keepers (==keeper_id)
```

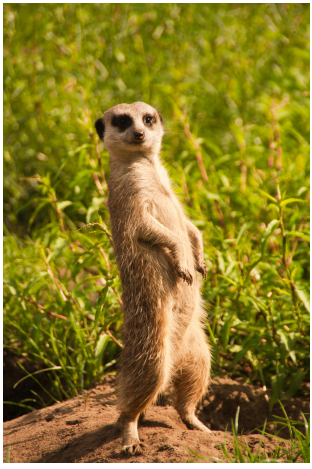
```
from animals
filter height_cm > 10
take 10
join k = keepers (==keeper_id)
filter k.first_name == "Thomas"
```

```
from animals
filter height_cm > 10
take 10
join k = keepers (==keeper_id)
filter k.first_name == "Thomas"
sort animals.age
```

```
...
take 10
join k = keepers (==keeper_id)
filter k.first_name == "Thomas"
sort animals.age
select {
    animals.age,
    species = animals.species,
    keeper = f"{k.first_name} {k.last_name}",
}
```

```
...
select {
    animals.age,
    species = animals.species,
    keeper = f"{k.first_name} {k.last_name}",
}
select {
    f"I'm {species}, {animals.age} years old",
    keeper
}
```

```
[
  {
    "I'm Suricata suricatta, 3 years old",
    keeper = "Thomas Edison"
  },
  {
    "I'm Capra ibex, 12 years old",
    keeper = "Thomas Moor"
  },
]
```




Arti
www.arcticphoto.is

- Top to bottom
- Easy exploration
- Extract a variable
- Extract a function

```
from animals
filter height_cm > 10
take 10
join k = keepers (==keeper_id)
filter k.first_name == "Thomas"
sort animals.age
select {
    animals.age,
    species = animals.species,
    keeper = f"{k.first_name} {k.last_name}",
}
select {
    f"I'm {species}, {animals.age} years old"
    keeper
}
```

Extract a variable

```
from animals
filter height_cm > 20
take 10
```

Extract a variable

```
let big_animals = (
    from animals
    filter height_cm > 20
)

from big_animals
take 10
```

Extract a function

```
from animals
sort {-height_cm}
take 5
```

Extract a function

```
let take_biggest = n rel -> (
    rel
    sort {-height_cm}
    take n
)

from animals
take_biggest 5
```

Data model

Basic data types

`bool`, `int`, `float`, `str`

Data model

Tuples

```
{my_int = 5, 4.2, my_bool = true}
```

- named fields
- different types
- static number of fields

Data model

Arrays

```
[1, 2, 10, -3]
```

- unnamed items
- items have the same type
- dynamic number of items

Data model

Relation $:=$ an array of tuples

```
[
    {my_int =  5, 4.2, my_bool = true},
    {my_int = -2, 6.1, my_bool = false},
    {my_int = 12, 3.0, my_bool = false},
]
```

Declarations

```
let a = 5

let b = a + 1
```

Functions

```
let add_one = x -> x + 1

let add = x y -> x + y
```

Functions

```
let five = (add_one 4)

let six = (add 4 2)
```

Functions

```
let seven = (5 | add_one | add_one)

let seven = (
    5
    add_one
    add_one
)
```

Transforms

Transform := a function on relations

```
let animals = ...

let main = (filter (height_cm > 20) animals)
```

Transforms

```
let animals = ...

let main = (animals | filter (height_cm > 20))
```

Transforms

```
let animals = ...

let main = (
    animals
    filter (height_cm > 10)
)
```

Transforms

```
let animals = ...

animals
filter (height_cm > 10)
```

Transforms

```
from animals
filter (height_cm > 10)
```

Transforms

```
from animals
filter height_cm > 10
```

Transforms

```
std.from
std.select
std.derive
std.filter
std.aggregate
std.sort
```

```
std.take
std.join
std.group
std.window
std.append
std.loop
```

Orthogonal

```
from expenses
filter dept == "Sales"
aggregate {total = sum cost}
filter total > 100.00
```

WHERE $\mapsto$ filter

HAVING $\mapsto$ filter

Orthogonal

Transform invariants:

- `filter` will not change columns

- `derive` & `select` will not change number of rows

- `aggregate` will produce exactly one row

Grouping

```
from expenses
aggregate {total = sum cost}
```

```
[
    {total = 431.22},
]
```

Grouping

```
from expenses
group dept (
    aggregate {total = sum cost}
)
```

```
[
  {dept = "Sales",       total = 331.00},
  {dept = "Accounting", total = 100.22},
]
```

Grouping

```
from expenses
group dept (
    take 1
)
```

```
[
  {dept = "Sales",       id = 33, cost =  5.30},
  {dept = "Accounting", id = 45, cost = 12.22},
]
```

Grouping

```
from expenses
group dept (
    sort {-cost}
    take 1
)

[
  {dept = "Sales",       id =  3, cost = 33.30},
  {dept = "Accounting", id = 16, cost = 12.22},
]
```

Grouping

```
from expenses
group expenses.* (
    take 1
)
```

Grouping

```
from expenses
group expenses.* (
    take 1
)
```

```
SELECT DISTINCT *
FROM expenses
```

Nulls

```
# PRQL
my_col == null

-- SQL
my_col IS NULL
```

Ergonomics

```
from employees
derive {
  age = @2023-01-31 - birth_date,
  full_name = f"{first_name} {last_name}",
  manager = reports_to ?? "No one",
  salary = 1_000_000,
#  is_fired = "No",
}
```

Challenges of

Compiling queries

SQL as a compilation target

How is this language executed?

Replace SQL as:

✗ database interface

✓ a query language

The task of a database interface

Imagine a database without a query language.

```
SELECT * FROM albums
```

... and then transform in client code.

→ super slow

The task of a database interface

Extreme example:

```
SELECT COUNT(*)
FROM albums
WHERE title LIKE 'The %'
```

The task of a database interface

Processing should be close to data

- minimal data transfer

- parallelism

- vectorization

The task of a database interface

Databases are:

- execution platforms

- compilation targets

Analogous to amd64, JVM

Dialects

Differences in:

- syntax (TOP vs LIMIT)

- available functions

- available data types

Dialects

Different:

- priorities

- backward compatibility guarantees

- implementation limitations

Dialects

No clear & robust specification

Compilers could:

- adapt query to target database

- produce error early

Leaky abstractions

Database interface should be transparent

Currently, this is not the case:

- invalid SQL

- sub-optimal SQL

- runtime errors

PRQL, the project

– an opensource effort

Licence

Apache License 2.0

Open community

Will never monetize

prqlc

compiler from PRQL to SQL

targets: sql.postgres, sql.sqlite, sql.duckdb, sql.mysql, sql.clickhouse

bindings for C, Python, JS, Java, .NET, PHP

prqlc

Don’t connect, infer

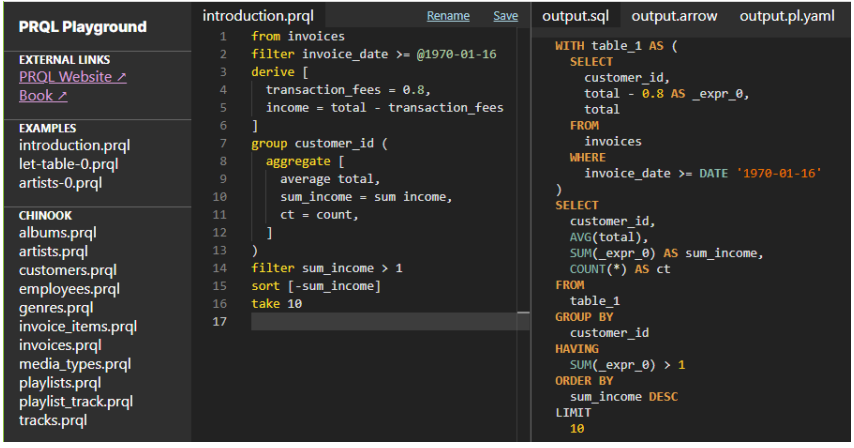
Fail early

```
Error:
   ╭─[:2:8]
   │
 2 │ select column_name = [track_id, name]
   ·        ─────────────┬────────────────
   ·                     └──────────────── unexpected assign to `column_name`
   ·
   · Help: move assign into the list: `[column_name = ...]`
───╯
```

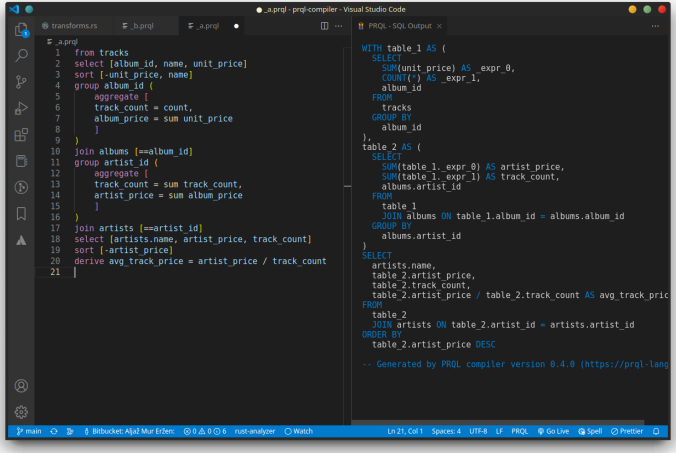
Architecture

PRQL $\rightarrow$ **PL** $\rightarrow$ **RQ** $\rightarrow$ **SQL**

Check it out: playground



Check it out: VSCode extension



Check it out

```
pip install pyprql
install.packages("prqlr")
npm install prql
cargo add prql-compiler
```

https://prql-lang.org

https://github.com/PRQL/prql

https://discord.gg/TfyM755m

Please vote and leave feedback!

Remember to vote and share feedback on the QCon App.

Any questions?

QCon by InfoQ