PRQL

Pipelined Relational Query Language

Aljaž Mur Eržen

Compiler developer @EdgeDB

```
from albums
filter album_id > 100
sort albums.title
take 10
join artists (==artist_id)
select {
    albums.album_id,
    albums.title,
    f"Artist name: {artist.name}",
}
```

Why?

There are transition costs!

Overview

Flaws of SQL

Language for relations

Compiling queries

PRQL, the project

A deef dive into

Flaws of SQL

Origins of the relational model

1970, Edgar F. Codd: abstraction over data storage

$\rightarrow$ Tuple relational calculus

1974, Donald D. Chamberlin & Raymond F. Boyce: SEQUEL

$\rightarrow$ Not a “proper” programming language

Not really composable

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

Not really composable

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

Patched syntax

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

Patched syntax

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

Patched syntax

```
SELECT DISTINCT name
FROM invoices
```

Patched syntax

```
SELECT EVALUATE TYPE-EMPLOYEE
  WHEN "F"
    MOVE "FULL TIME" TO EMP-TYPE-PR
  WHEN "P"
    MOVE "PART TIME" TO EMP-TYPE-PR
  WHEN "C"
    MOVE "CONSULTANT" TO EMP-TYPE-PR
  WHEN OTHER
    MOVE "INVALID" TO EMP-TYPE-PR
```

Patched syntax

Too much syntax

... but also ...

Not enough syntax

Name resolution

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

Name resolution

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

Name resolution

More rules:

– ORDER BY positionals

– Correlated subqueries

– LATERAL

Relations vs scalars

```
SELECT * FROM table

SELECT count(*) FROM table
```

Relations vs scalars

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

Relations vs scalars

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

Relations cannot be ordered

```
SELECT * FROM albums ORDER BY title
```

Relations cannot be ordered

```
SELECT
  *,
  ... AS my_col
FROM (
    SELECT * FROM albums ORDER BY title
) inner
```

Relations cannot be ordered

```
SELECT
  *,
  ROW_NUMBER()
    OVER (ORDER BY artist_id) AS my_col
FROM (
    SELECT * FROM albums ORDER BY title
) inner
```

Relations cannot be ordered

SELECT returns an **ordered set**

FROM pulls-in a **set**

Relations cannot be ordered

```
SELECT
  *,
  ... AS my_col
FROM (
    SELECT *
    FROM albums
) inner
ORDER BY title
```

Relations cannot be ordered

```
SELECT
  *,
  ... AS my_col
FROM (
    SELECT * FROM albums ORDER BY title LIMIT 10
) inner
ORDER BY title
```

Identity of aggregation

```
SELECT SUM(cost) FROM expenses WHERE FALSE
```

Two possible behaviors: NULL or 0

Both valid

Identity of aggregation

“Every marble in this bag is black”
… but the bag is empty.

Ancient greeks say FALSE

Modern logic says TRUE

SQL says NULL

Identity of aggregation

Homomorphism of addition

```
SUM([1]) + SUM([4, 5]) = SUM([1, 4, 5])
```

```
1 + 9 = 10
```

Identity of aggregation

```
SUM([1]) + SUM([]) = SUM([1])

       1 +    ?    = 1

        -> SUM([]) = 0
```

identity of addition

Identity of aggregation

```
COUNT([])       = 0
ARRAY_AGG([])   = []
SUM([])         = 0
ANY([])         = false
EVERY([])       = true
STRING_AGG([]) = ''
```

Dialects

Differences in:

- syntax (TOP vs LIMIT)

- available functions

- available data types

Dialects

A class of languages

There is a standard

Slight deviations

Dialects

Different:

- priorities

- backward compatibility guarantees

- implementation limitations

Dialects

No clear & robust specification

Compilers could:

- adapt query to target database

- produce error early

Design of a new

Language for relations

Tuple relational calculus

Relation $\sim$ a set of tuples

$$\pi_{track\_id,name,title}(R)$$

$$\sigma_{track\_id=5}(R)$$

$$R * S$$

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

Relations $\sim$ 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

```
let invoices = ...

let main = (filter (total > 10) invoices)
```

Transforms

```
let invoices = ...

let main = (invoices | filter (total > 10))
```

Transforms

```
let invoices = ...

let main = (
    invoices
    filter (total > 10)
)
```

Transforms

```
let invoices = ...

invoices
filter (total > 10)
```

Transforms

```
from invoices
filter (total > 10)
```

Transforms

```
from invoices
filter total > 10
```

Top to bottom

```
from albums
filter album_id > 100
sort albums.title
```

Top to bottom

```
from albums
filter album_id > 100
sort albums.title
take 10
```

Top to bottom

```
from albums
filter album_id > 100
sort albums.title
take 10
join artists (==artist_id)
```

Top to bottom

```
from albums
filter album_id > 100
sort albums.title
take 10
join artists (albums.artist_id == artists.artist_id)
```

Top to bottom

```
from albums
filter album_id > 100
sort albums.title
take 10
join artists (==artist_id)
select {
    albums.album_id,
    albums.title,
    f"Artist name: {artist.name}",
}
```

Top to bottom

- Convenient for exploration
- Lazy evaluation
- Extract a variable
- Extract a function

Top to bottom

```
let take_cheapest = n rel -> (
    rel
    sort unit_price
    take n
)

from tracks
take_cheapest 5
```

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 = 33, cost = 5.30},
  {dept = "Accounting", id = 16, cost = 1.22},
]
```

Grouping

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

Grouping

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

```
SELECT DISTINCT *
FROM expenses
```

Nulls

```
# PRQL
null == null  # true

my_col == null

-- SQL
my_col IS NULL
```

Micro-features

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

Challenges of

Compiling queries

SQL as a compilation target

How is this language executed?

✗ database interface

✓ a query language

The task of a query lanuage

Imagine a database without a query language.

```
SELECT * FROM albums
```

... and then transform in client code.

$\rightarrow$ super slow

The task of a query lanuage

Extreme example:

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

The task of a query lanuage

Processing should be close to data

- minimal data transfer

- parallelism

- vectorization

The task of a query lanuage

Databases are:

- execution platforms

- compilation targets

Analogous to amd64, JVM

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

The compiler and its IRs

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

The compiler and its IRs

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

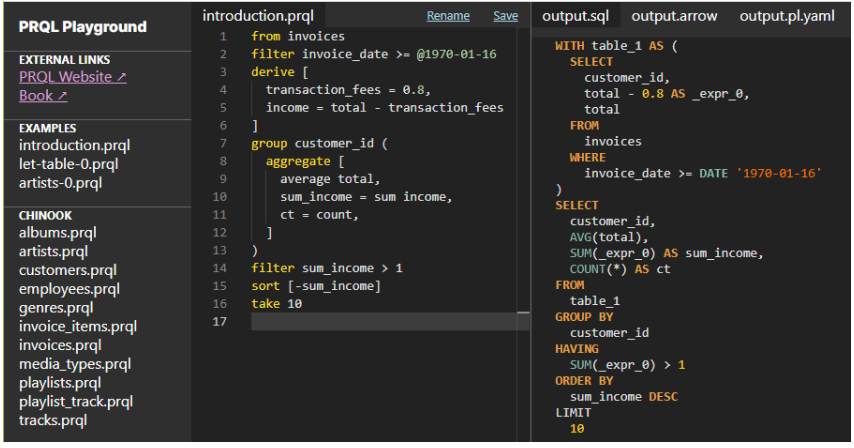
Licence

Apache

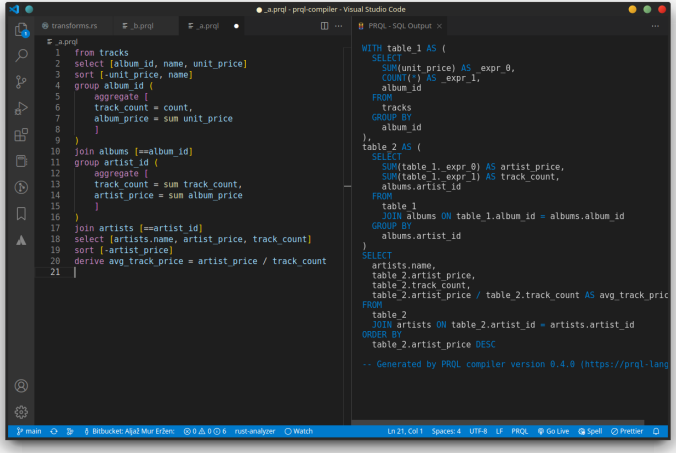
Open community

No plans to monetize

Check it out: playground



Check it out: VSCode extension



Check it out: prql-query - pq

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