

# PRQL

## Pipelined Relational Query Language

Aljaž Mur Eržen

Compiler developer @EdgeDB



```
from animals
filter height_cm > 20
take 10
join keepers (==keeper_id)
filter keepers.first_name == "John"
sort animals.age
select {
    species = animals.species,
    keeper = keepers.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 of the relational model

1970, Edgar F. Codd: abstraction over data storage

→ Tuple relational calculus

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
```



# Human-friendly 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
```

# 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

# 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)
```

# Relations vs scalars

```
SELECT * FROM table
```

```
SELECT count(*) FROM table
```

# Relations vs scalars

Depends on the database contents:

```
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'
)
```

Design of a new

# Language for relations

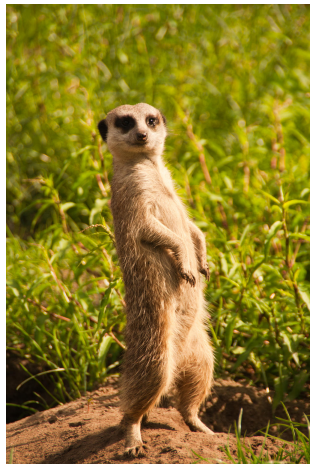
from animals

```
[  
  {  
    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 == "John"
```

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

...

take 10

join k = keepers (==keeper\_id)

filter k.first\_name == "John"

sort animals.age

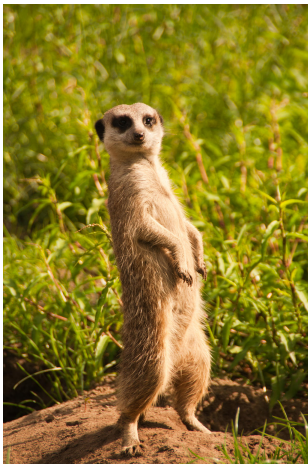
```
select {  
  animals.age,  
  species = animals.species,  
  keeper = f"{k.first_name} {k.last_name}",  
}
```

...

```
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 = "John Doe"  
  },  
  {  
    "I'm Capra ibex, 12 years old",  
    keeper = "John Doe"  
  },  
]
```





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

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

# Tuple relational calculus

Relation  $\sim$  a set of tuples

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

$$R * S$$

# 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

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  
null == null  # true
```

```
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

# 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 → PL → RQ → SQL**

# Licence

Apache License 2.0

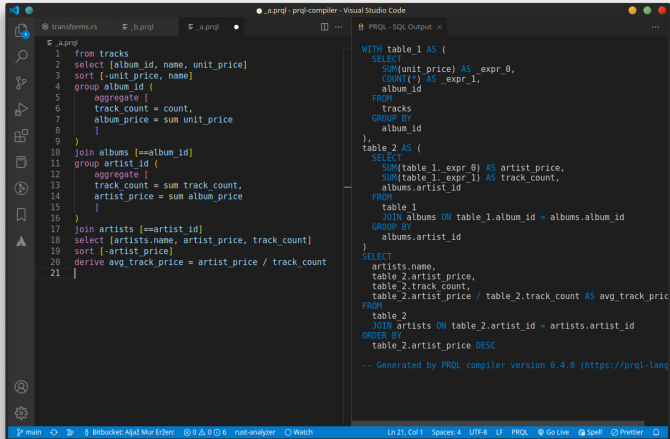
Open community

Will never monetize

# Check it out: playground

PRQL Playground	introduction.prql	<a href="#">Rename</a>	<a href="#">Save</a>	output.sql	output.arrow	output.pl.yaml
<b>EXTERNAL LINKS</b> <a href="#">PRQL Website ↗</a> <a href="#">Book ↗</a>	<pre>1 from invoices 2 filter invoice_date &gt;= @1970-01-16 3 derive [ 4   transaction_fees = 0.8, 5   income = total - transaction_fees 6 ] 7 group customer_id ( 8   aggregate [ 9     average total, 10    sum_income = sum income, 11    ct = count, 12  ] 13 ) 14 filter sum_income &gt; 1 15 sort [-sum_income] 16 take 10 17</pre>			<pre>WITH table_1 AS (   SELECT     customer_id,     total - 0.8 AS _expr_0,     total   FROM     invoices   WHERE     invoice_date &gt;= DATE '1970-01-16' ) SELECT   customer_id,   AVG(total),   SUM(_expr_0) AS sum_income,   COUNT(*) AS ct FROM   table_1 GROUP BY   customer_id HAVING   SUM(_expr_0) &gt; 1 ORDER BY   sum_income DESC LIMIT   10</pre>		
<b>EXAMPLES</b> introduction.prql let-table-0.prql artists-0.prql						
<b>CHINOOK</b> albums.prql artists.prql customers.prql employees.prql genres.prql invoice_items.prql invoices.prql media_types.prql playlists.prql playlist_track.prql tracks.prql						

# Check it out: VSCode extension



The screenshot shows the Visual Studio Code interface with the PRQL compiler extension. The left pane displays a PRQL file named `_a.prql` with the following code:

```
1 from tracks
2 select [album_id, name, unit_price]
3 sort [-unit_price, name]
4 group album_id (
5   aggregate [
6     track_count = count,
7     album_price = sum unit_price
8   ]
9 )
10 join albums [==album_id]
11 group artist_id (
12   aggregate [
13     track_count = sum track_count,
14     artist_price = sum album_price
15   ]
16 )
17 join artists [==artist_id]
18 select [artists.name, artist_price, track_count]
19 sort [-artist_price]
20 derive avg_track_price = artist_price / track_count
21
```

The right pane shows the generated SQL in a file named `PRQL - SQL Output`:

```
WITH table_1 AS (
  SELECT
    SUM(unit_price) AS _expr_0,
    COUNT(*) AS _expr_1,
    album_id
  FROM
    tracks
  GROUP BY
    album_id
),
table_2 AS (
  SELECT
    SUM(table_1._expr_0) AS artist_price,
    SUM(table_1._expr_1) AS track_count,
    albums.artist_id
  FROM
    table_1
  JOIN albums ON table_1.album_id = albums.album_id
  GROUP BY
    albums.artist_id
)
SELECT
  artists.name,
  table_2.artist_price,
  table_2.track_count,
  table_2.artist_price / table_2.track_count AS avg_track_price
FROM
  table_2
JOIN artists ON table_2.artist_id = artists.artist_id
ORDER BY
  table_2.artist_price DESC

-- Generated by PRQL compiler version 0.4.0 (https://prql-lang
```

The status bar at the bottom indicates the file is `main`, the encoding is `UTF-8`, and the language is `PRQL`. It also shows various extensions like `Go Live`, `Spell`, and `Prettier` are active.



# Check it out: prql-query - pq

```
chinook$ pq --from tracks.csv 'select [track_id, name, bytes] | take 10'
```

track_id	name	bytes
1	For Those About To Rock (We Salute You)	11170334
2	Balls to the Wall	5510424
3	Fast As a Shark	3990994
4	Restless and Wild	4331779
5	Princess of the Dawn	6290521
6	Put The Finger On You	6713451
7	Lets Get It Up	7636561
8	Inject The Venom	6852860
9	Snowballed	6599424
10	Evil Walks	8611245

```
chinook$
```

# 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?