Preql Documentation

Erez Shinan

Jun 13, 2022

Overview

1 Preface 3

2 Preql 5
- 2.1 Preql compiles to SQL . . . 5
- 2.2 Preql is interpreted . . . 5
- 2.3 Better syntax, semantics, and practices . . . 6
- 2.4 Escape hatch to SQL . . . 6

3 Conclusion 7

4 Features 9
- 4.1 Planned features . . . 10

5 Performance 11
- 5.1 Components . . . 11
- 5.2 Benchmarks . . . 11

6 Roadmap 13

7 FAQ 15
- 7.1 Technical Help . . . 15
- 7.2 Community and Support . . . 15
- 7.3 License . . . 16

8 Tutorial for the basics of Preql 17
- 8.1 What is Preql? . . . 17
- 8.2 Getting Started (Install & How to use) . . . 17
- 8.3 Basic Expressions . . . 18
- 8.4 Functions . . . 20
- 8.5 Tables . . . 21
- 8.6 The SQL Escape-hatch . . . 28
- 8.7 Notable Built-in functions . . . 29
- 8.8 Calling Preql from Python . . . 29

9 Code comparison: Preql, SQL and the rest 31
- 9.1 Table Operations . . . 31
- 9.2 Gotchas . . . 34
- 9.3 Programming . . . 34

10 Getting Started 37
10.1 Install . . . 37
10.2 Run the interpreter in the console (REPL) . . . 37
10.3 Run in a Jupyter Notebook . . . 38
10.4 Use as a Python library . . . 38
10.5 Run as a REST / GraphQL server . . . 39
10.6 Further reading . . . 39

11 Why not SQL/ORM/Pandas ? 41
11.1 SQL . . . 41
11.2 ORMs . . . 42
11.3 Pandas . . . 42

12 Language Reference 43
12.1 Literals . . . 43
12.2 Keywords . . . 45

13 Preql Types 47

14 Preql Modules 51
14.1 __builtins__ . . . 51
14.2 graph . . . 64

15 Python API Reference 67
15.1 Preql . . . 67

16 Resources 69

Index 71

Preql

CHAPTER 1

Preface

Relational databases are a common and powerful approach to storing and processing information. Based on the solid foundation of relational algebra, they are efficient, resilient, well-tested, and full of useful features.

However, they all share the same weakness: They all use an antiquated programming interface called SQL.

While SQL was clever and innovative at the time of its conception, today we can look back on its design and see it has many fundamental mistakes, which make SQL incompatible with our contemporary notions of how a programming language should look and work.

As data becomes ever more important to the world of computation, so grows the appeal of a better solution. This need for an alternative inspired us to create the Preql programming language.

CHAPTER 2

Preql

Preql is a new programming language that aims to replace SQL as the standard language for programming databases.

These are big shoes to fill. Here is how Preql intends to do it:

2.1 Preql compiles to SQL

Like SQL, Preql is guided by relational algebra, and is designed around operations on tables.

In Preql, `table` is a built-in type that represents a database table, and all table operations, such as filtering, sorting, and group-by, are compiled to SQL.

That means Preql code can run as fast and be as expressive as SQL.

Preql supports multiple targets, including PostgreSQL, MySQL and SQLite. See *features* for a complete list.

2.2 Preql is interpreted

Not everything can be done in SQL. Control-flow constructs like for-loops, or downloading a remote JSON file, aren't possible in every database implementation.

Some things, such as first-class functions, or reflection, aren't possible at all.

Whenever your Preql code can't be compiled to SQL, it will be interpreted instead.

Being interpreted also lets Preql adopt advanced concepts, like "everything is an object", support for `eval()`, and so on.

That means in Preql you can do anything you could do in Python or Javascript, even when SQL can't.

2.3 Better syntax, semantics, and practices

Preql's syntax is inspired by javascript. It's relatively concise, and familiar to programmers.

It integrates important ideas like Fail-Early, and the Principle Of Least Astonishment.

The code itself is stored in files, instead of a database, which means it can be version-controlled (using git or similar)

Preql also comes with an interactive prompt with auto-complete.

2.4 Escape hatch to SQL

There are many dialects of SQL, and even more plugins and extensions. Luckily, we don't need to support all of them.

Preql provides the builtin function SQL(), which allows you to run arbitrary SQL code, anywhere in your Preql code.

CHAPTER 3

Conclusion

Preql's design choices allow it to be fast and flexible, with a concise syntax.

Preql adopts the good parts of SQL, and offers improvements to its weakest points.

Where to go next?

- *Read the tutorial* as it takes you through the basics of Preql
- Browse the examples and learn how Preql can be used.

CHAPTER 4

Features

Preql is a programming language, a library, an interactive shell, and a set of tools.

- **Modern syntax and semantics**
  - Interpreted, everything is an object
  - Strong type system with gradual type validation and duck-typing
  - Modules, functions, exceptions, tables, structs
- **SQL integration**
  - Compiles to SQL whenever possible (guaranteed for all table operations)
  - Escape hatch to SQL (write raw SQL expression within Preql code)
  - Support for multiple SQL targets
    - **SQLite**
    - **PostgreSQL**
    - **MySQL**
    - **BigQuery**
    - Askgit :)
    - More to come!
- **Python integration**
  - Use from Python as a library
  - Call Python from Preql
  - Pandas integration
- **Interactive Environment**
  - Shell (REPL), with auto-completion
  - Runs on Jupyter Notebook, with auto-completion

- Thread-safe
- REST+JSON server, automatically generated

4.1 Planned features

- See the *roadmap*

CHAPTER 5

Performance

5.1 Components

For understanding performance, Preql should be considered as split into two components:

In a typical Preql program, most of the time will be spent in executing SQL, rather than in the interpreter.

Running SQL through Preql adds a constant-time cost, due to real-time compilation. This may be noticeable in very fast queries, such as fetching a single row by id.

Future versions of Preql will cut the constant-time cost significantly, by caching the compiled SQL (a poc is already working).

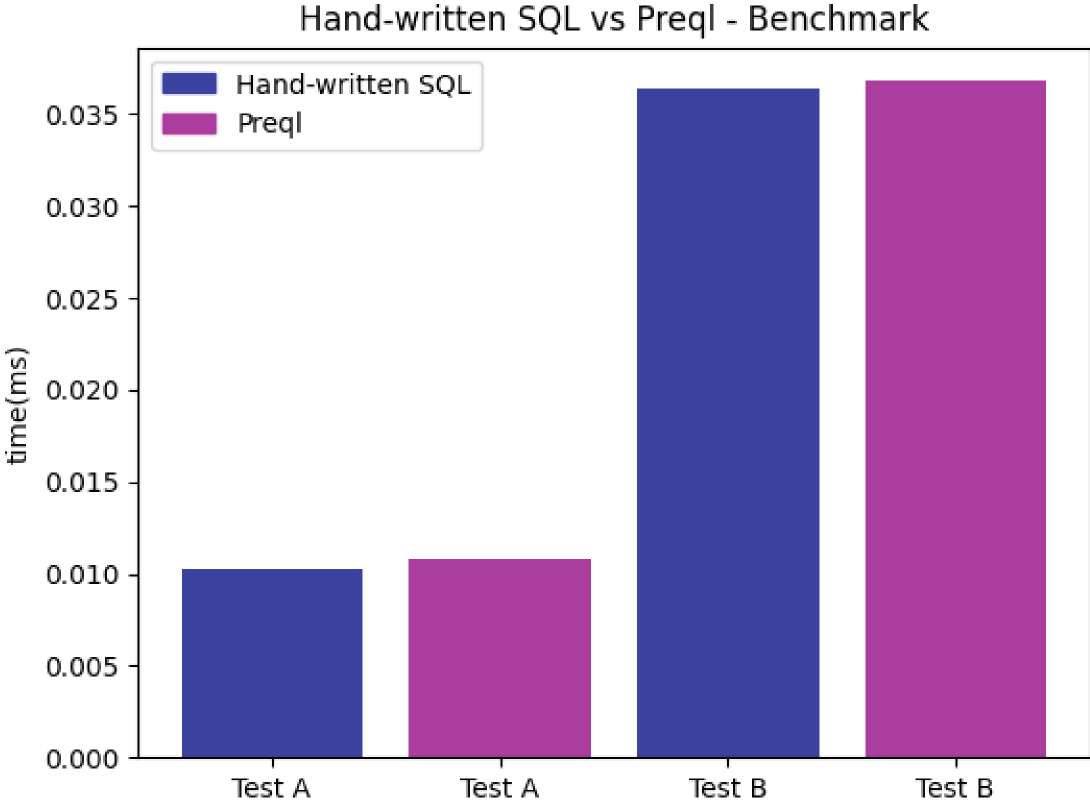
5.2 Benchmarks

5.2.1 Comparison to hand-written SQL

The following benchmark compared the performance of hand-written SQL queries to Preql-generated SQL (without compilation costs).

The code is available on benchmark/test_chinook.py

Results for Sqlite3 (1000 iterations):



- Test A - Simple selection and projection
- Test B - Multiple joins and a groupby

CHAPTER 6

Roadmap

Preql still has a long way to go, in terms of features and capabilities.

While plans can always change, here is a list of features that will probably make it in the near future:

- **Language**
  - JSON operations in SQL
  - API for Graph computation over SQL
  - Multiple Dispatch (multimethods)
  - Automatic joins via attribute access
  - Automatic Many-to-many
- **Compilation Technology**
  - JIT compilation (PoC is already working!)
  - Compile control flow to SQL
- **Support for more databases**
  - Redshift
  - CockroachDB
  - TimeSeries
  - MongoDB?
  - More…
- **Usability and integration**
  - IDE support (vscode, etc.)
  - Automatically generate GraphQL interface
  - Migrations

CHAPTER 7

FAQ

7.1 Technical Help

7.1.1 I'm new to Python. How do I install Preql?

First, you need to make sure Python is installed, at version 3.6 or above. You can get it from https://www.python.org/downloads/.

Then, open your command-line or shell, and write the following:

```
python -m pip install preql --user
```

The Python executable might be called `python3`, or `python3.9` etc.

The `--user` switch ensures you won't need special permissions for the installation.

7.2 Community and Support

7.2.1 I think I found a bug. What do I do?

We'll do our best to solve bugs as quickly as possible.

If you found a bug in Preql, open an issue here: https://github.com/erezsh/Preql/issues/new

Include any information you can share. The best way is with a minimal code example that runs and demonstrates the error.

7.2.2 Where can I ask questions?

You can ask any question here: https://github.com/erezsh/Preql/discussions

7.2.3 Contact Me

If you want to contact me privately, you may do so through email, at erezshin at gmail.com, or through twitter.

I am also available for paid support.

7.3 License

7.3.1 Can I use Preql in my project/company/product?

Preql is completely free for personal use, including internal use in commercial companies.

For use in projects and products, as a library, the license differentiates between two kinds of use:

1. Projects or products that use Preql internally, and don't expose the language to the user, may consider Preql as using the MIT license. That also applies to commercial projects and products.
2. Projects or products that intentionally expose the language to the user, as part of their interface. Such use is only allowed for non-commercial projects, and then they must include the Preql license.

If you would like to embed the Preql language in your commercial project, and to benefit from its interface, contact us to buy a license.

7.3.2 Why not dual license with GPL, AGPL, or other OSI-approved license?

In the history of open-source, GPL and AGPL were often used as a subtle strategy to disuade unfair commercial use. Most companies, and especially corporations, didn't want to share their own code, and so they had to buy a license if they wanted to use it.

Unfortunately, GPL and even AGPL don't fully protect software from exploitation by competitors, particularly cloud providers.

That is why many open-source projects, who were once AGPL, BSD or Apache 2.0, have decide to start using to their own license. Famous examples include Redis, Confluent, MongoDB, and Elastic.

7.3.3 Is Preql open-source?

Preql's license is in line with some definitions of "open source", but does not fit the definition outlined by the OSI.

For practical purposes, Preql can be called "source available", or transparent-source.

CHAPTER 8

Tutorial for the basics of Preql

8.1 What is Preql?

Preql is a relational programming language that runs on relational databases, such as PostgreSql, MySql, and Sqlite.

It does so by compiling to SQL. However, its syntax and semantics resemble those of Javascript and Python.

By combining these elements, Preql lets you write simple and elegant code that runs as fast as SQL.

We'll soon dive into the language itself, but first let's install and learn how to use the interpreter

8.2 Getting Started (Install & How to use)

You can install preql by running this in your shell/command prompt:

```
$ pip install -U preql
```

Usually, you would connect Preql to a database, or load an existing module.

But, you can also just run the preql interpreter as is:

```
$ preql
Preql 0.1.16 interactive prompt. Type help() for help
>>
```

By default, the interpreter uses SQLite's memory database. We'll later see how to change it using the `connect()` function.

From now on, we'll use >> to signify the Preql REPL.

Press Ctrl+C at any time to interrupt an existing operation or prompt. Press Ctrl+D or run `exit()` to exit the interpreter.

You can run the `names()` function to see what functions are available, and `help()` to get interactive help.

You can also run a preql file. Let's create a file called `helloworld.pql`:

```
// helloworld.pql
print "Hello World!"
```

And then run it:

```
$ preql -m helloworld
Hello World!
```

Alternatively, we could do `preql -f helloworld.pql`, if we want to specify a full path.

We can also use Preql as a Python library:

```
# In the Python interpreter
from preql import Preql
p = Preql()

assert p('sum([1..10])') == 45

p('''
  func my_range(x) = [1..x]
''')
print(p.my_range(8))
# Output:
# [1, 2, 3, 4, 5, 6, 7]
```

8.3 Basic Expressions

Preql has integers, floats and strings, which behave similarly to Python

`null` behaves just like Python's None.

```
>> 1 + 1
2
>> 2 / 4
0.5
>> 27 % 13
1
>> "a" + "b"
"ab"
>> "-" * 5
"-----"

>> (not 0) and 2 < 4
True
>> null == null     // Unlike SQL!
True
>> null + 1
Exception traceback:
  ~~~ At '<repl>' line 1, column 6
null + 1
-----^---

TypeError: Operator '+' not implemented for nulltype and int
```

Notice that dividing two integers results in a float. To get an integer, use the /~ operator, which is equivalent to Python's // operator:

```
>> 10 /~ 3
3
```

You can get the type of anything in Preql by using the `type()` function:

```
>> type(10)
int
>> type(int)
type
```

Preql also has lists, which are essentially a table with a single column called `item`:

```
 >> my_list = [1,2,3]
 table
   =3

 item

│     1 │
│     2 │
│     3 │
└──────
 >> count(my_list + [4,5,6])
6
 >> names(my_list)
       table  =1

 name  type  doc

│ item │ int  │      │
└─────────────
 >> type(my_list)
list[int]
 >> type(["a", "b", "c"])
list[string]
```

The range syntax creates a list of integers:

```
>> [1..100]
table  =99

   item

│       1 │
│       2 │
│       3 │
│       4 │
│       5 │
│     ... │
└───────
```

Preql only shows us a preview of the table. If we want to see more items, we can just enter a dot (`.`) in the prompt:

```
>> .
table [5..] =99
```

(continues on next page)

(continued from previous page)

```
  item
 ┌─────────┐
 │      6  │
 │      7  │
 │      8  │
 │      9  │
 │     10  │
 │    ...  │
 └─────────┘
```

Entering `.` again will keep scrolling more items.

8.3.1 inspect_sql

You might be curious what SQL statements are being executed behind the scenes. You can find out using the `inspect_sql()` function.

```
>> print inspect_sql([1..10] + [20..30])
```

```
WITH RECURSIVE range1 AS (SELECT 1 AS item UNION ALL SELECT item+1 FROM range1 WHERE␣
↪item+1<10)
    , range2 AS (SELECT 20 AS item UNION ALL SELECT item+1 FROM range2 WHERE item+1
↪<30)
    SELECT * FROM [range1] UNION ALL SELECT * FROM [range2] LIMIT -1
```

8.4 Functions

Declare functions using func:

```
func sign(x) {
  if (x == 0) {
    return 0
  } else if (x > 0) {
    return 1
  } else {
    return -1
  }
}
>> sign(-100)
-1
>> sign(100)
1
```

You can also use them in table operations!

```
>> [-20, 0, 30]{ sign(item) }
table
  =3

sign
```

(continues on next page)

(continued from previous page)

```
│   -1 │
│    0 │
│    1 │
└──────┘
```

Let's inspect the SQL code that is executed:

```
>> print inspect_sql([-20, 0, 30]{ sign(item) })
```

```
WITH RECURSIVE list_1([item]) AS (VALUES (-20), (0), (30))
    SELECT CASE WHEN ([item] = 0) THEN 0 ELSE CASE WHEN ([item] > 0) THEN 1 ELSE -1␣
↪END  END  AS [sign] FROM [list_1]
```

Note: Functions with side-effects or I/O operations aren't allowed in table operations, due to SQL's limitations.

There's also a shorthand for "one-liners":

```
>> func str_concat(s1, s2) = s1 + s2
>> str_concat("foo", "bar")
"foobar"
```

Functions are objects just like everything else, and can be passed around to other functions.

Here is a toy example that demonstrates this:

```
func apply_function(f, x) = f(x)

my_list = ["this", "is", "a", "list"]

// Run `apply_function` for each item, and use the built-in `length` function for␣
↪strings.
// `len` is just the name of the new column.
print my_list{
  len: apply_function(length, item)
}
// Output:
// table  =4
//
//    len
//
// │      4 │
// │      2 │
// │      1 │
// │      4 │
// └───────┘
```

8.5 Tables

Tables are essentially a list of rows, where all the rows have the same structure.

That structure is defined by a set of columns, where each column has a name and a type.

Preql's tables are stored in an SQL database, and most operations on them are done efficiently using SQL queries.

Here is how we would define a table of points:

```
table Point {
    x: float
    y: float
}
```

This statement creates a persistent table named `Point` in your database (if you are connected to one. The default database resides in memory and isn't persistent). The executed SQL looks like this:

```
CREATE TABLE IF NOT EXISTS "Point" ("id" INTEGER, "x" FLOAT NOT NULL, "y" FLOAT NOT NULL, PRIMARY KEY (id))
```

If the table `Point` already exists, it will instead verify that the new definition is a subset of the existing one. That is, that all the columns defined in it exist in the current table, and with the correct type.

For this tutorial, let's create a table that's little more meaningful, and populate it with values:

```
table Country {
  name: string
  population: int
}

palau = new Country("Palau", 17900)
nauru = new Country("Nauru", 11000)
new Country("Tuvalu", 10200)
```

`new` accepts its parameters in the order that they were defined in the table. However, it's also possible to use named arguments, such as `new Point(y:10, x:1)`.

In the above example, we assigned the newly created rows to variables. But they also exist independently in the table.

We can see that the `Country` table has three rows:

```
>> count(Country)
3
```

The `new` statements inserted our values into an SQL table, and `count()` ran the following query: `SELECT COUNT(*) FROM Country`

(Note: You can see every SQL statement that's executed by starting the REPL with the `--print-sql` switch.)

We can also observe the variables, or the entire table:

```
 >> palau
Row{id: 1, name: "Palau", population: 17900}
>> palau.population + 1
17901
>> Country
      table Country =3

 id  name    population

 │  1 │ Palau  │       17900 │
 │  2 │ Nauru  │       11000 │
 │  3 │ Tuvalu │       10200 │
 └──────────────────────────
```

Notice that every table automatically gets an `id` column. It's a useful practice, that provides us with an easy and performant "pointer" to refer to rows.

8.5.1 Table operations

There are many operations that you can perform on a table. Here we'll run through the main ones.

Selection lets us filter tables using the selection operator:

```
// All countries that contain the letter 'l' and a population below 15000
>> Country[name like "%l%", population < 15000]

 id  name    population

│  3 │ Tuvalu │      10200 │
└─────────────────────────
```

We can chain table operations:

```
 >> Country[name like "%l%" or population < 11000] {name, population}
       table  =2

 name    population

│ Palau  │      17900 │
│ Tuvalu │      10200 │
└─────────────────────
```

We can also filter the rows by index (zero-based), by providing it with a `range` instead.

```
 >> Country[1..]
      table Country =2

 id  name    population

│  2 │ Nauru  │      11000 │
│  3 │ Tuvalu │      10200 │
└─────────────────────────
```

Notice that the row index and the value of the `id` column are not related in any meaningful way.

Projection lets us create new tables, with columns of our own choice:

```
>> Country{name, is_big: population>15000}
     table  =3

 name    is_big

│ Palau  │      1 │
│ Nauru  │      0 │
│ Tuvalu │      0 │
└─────────────────

>> Country[name like "P%"]{name, is_big: population>15000}
    table  =1

 name   is_big

│ Palau │      1 │
└────────────────

>> func half(n) = n / 2
>> Country{..., half(population)}   // Ellipsis fills in the rest of the columns
```

(continues on next page)

(continued from previous page)

```
              table  =3

 id  name    population    half

 │  1 │ Palau  │      17900 │ 8950.0 │
 │  2 │ Nauru  │      11000 │ 5500.0 │
 │  3 │ Tuvalu │      10200 │ 5100.0 │
 └────────────────────────────────
```

Notice that Preql creates a new table type for each projection. Therefore, the fields that aren't included in the projection, won't be available afterwards.

However these are only types, and not actual tables. To create a persistent table, we can write:

```
table half_population = Country{..., half(population)}
```

Now, if connected to a database, `half_population` will be stored persistently.

Aggregation looks a lot like projection, and lets us aggregate information:

The syntax is basically `{ keys => values }`

```
// Count how many countries there are, for each length of name.
>> Country { length(name) => count(id) }
    table  =2

 length  count

 │      5 │      2 │
 │      6 │      1 │
 └───────────────


// If no keys are given, aggregate all the rows into one.
>> world_population = Country { => sum(population) }
table  =1

   sum

 │ 39100 │
 └───────


// We can extract the row from the table using the `one` operator
>> one world_population
Row{sum: 39100}

// Create an even-odd histogram
>> [1,2,3,4,5,6,7] {
      odd: item % 2 => count(item)
   }
   table  =2

 odd  count

 │   0 │      3 │
 │   1 │      4 │
 └────────────


// Sum up all the squares
```

(continues on next page)

(continued from previous page)

```
>> func sqrsum(x) = sum(x*x)
>> [1,2,3,4]{ => sqrsum(item)}
table  =1

 sqrsum

│      30 │
└────────
```

Ordering lets us sort the rows into a new table.

```
>> Country order {population} // Sort ascending
      table Country =3

 id  name    population

│  3 │ Tuvalu │       10200 │
│  2 │ Nauru  │       11000 │
│  1 │ Palau  │       17900 │
└─────────────────────────

>> Country order {^name}      // Sort descending (^)
      table Country =3

 id  name    population

│  3 │ Tuvalu │       10200 │
│  1 │ Palau  │       17900 │
│  2 │ Nauru  │       11000 │
└─────────────────────────
```

8.5.2 Lazy-evaluation vs Temporary tables

Immutable table operations, such as selection and projection, are lazily-evaluated in Preql. That means that they don't execute until strictly necessary.

This allows for gradual chaining, that the compiler will then know to merge into a single query:

```
a = some_table[x > 100]    // No SQL executed
b = a {x => sum(y)}        // ... same here
first20 = b[..20]          // ... same here
print first20              // Executes a single SQL query for all previous statements
print first20              // Executes the same SQL query all over again.
```

Lazy-evaluation for queries has the following advantages:

- Results in better compilation
- Leaner memory use, since we don't store intermediate results
- The results of the query are 'live', and update whenever the source table updates.

However, in situations when the same query is used in several different statements, it may be inefficient to run the same query again and again.

In those situations it may be useful to store the results in a temporary table:

```
table first20 = b[..20]    // Execute a single SQL query and store it
print first20              // Only needs to query the 'first20' table
print first20              // Only needs to query the 'first20' table
```

A temporary table is a table that's persistent in the database memory for as long as the session is alive.

Here's another example:

```
// Create a temporary table that resides in database memory
>> table t_names = Country[population>100]{name}  // Evaluated here once
>> count(t_names) + count(t_names)

// Create a query through lazy-evaluation. It's just a local definition
>> q_names = Country[population>100]{name}
>> count(q_names) + count(q_names)                // Evaluated here twice
```

The main disadvantage of using temporary tables is that they may fill up the database memory when working with large tables.

8.5.3 Update

We can **update** tables in-place.

Updates are evaluated immediately. This is true for all expressions that change the global state.

Example:

```
>> Country update {population: population + 1}
     table Country =3

 id  name    population

 │  1 │ Palau  │       17901 │
 │  2 │ Nauru  │       11001 │
 │  3 │ Tuvalu │       10201 │
 └─────────────────────────

 >> Country[name=="Palau"] update {population: population - 1}
     table Country =1

 id  name   population

 │  1 │ Palau │       17900 │
 └────────────────────────
 >> Country
      table Country =3

 id  name    population

 │  1 │ Palau  │       17900 │
 │  2 │ Nauru  │       11001 │
 │  3 │ Tuvalu │       10201 │
 └─────────────────────────
```

8.5.4 Join

Joining two tables means returning a new table that contains the rows of both tables, matched on a certain attribute.

It is possible to omit the attributes when there is a predefined relationship between the tables.

```
// Create tables from lists. That automatically adds an `id` column.
>> table odds = [1, 3, 5, 7, 9, 11]
>> table primes = [2, 3, 5, 7, 11]

// Join into columns `o` and `p`, which are structures containing the original rows.
>> join(o: odds.item, p: primes.item)

 o                       p

│ {'item': 3, 'id': 2}  │ {'item': 3, 'id': 2}  │
│ {'item': 5, 'id': 3}  │ {'item': 5, 'id': 3}  │
│ {'item': 7, 'id': 4}  │ {'item': 7, 'id': 4}  │
│ {'item': 11, 'id': 6} │ {'item': 11, 'id': 5} │
└───────────────────────────────────────────────

// We can then destructure it into a regular table
>> join(o: odds.item, p: primes.item) {o.item, o_id: o.id, p_id: p.id}
      table  =4

 item  o_id  p_id

│    3 │    2 │    2 │
│    5 │    3 │    3 │
│    7 │    4 │    4 │
│   11 │    6 │    5 │
└──────────────────

// We can filter countries by name, by joining on their name:
>> join(c: Country.name, n:["Palau", "Nauru"].item) {...c}
         table  =2

 id  name   population

│  1 │ Palau │       17900 │
│  2 │ Nauru │       11001 │
└────────────────────────

// But idiomatically, the best way to accomplish this is to use the `in` operator
>> Country[name in ["Palau", "Nauru"]]
     table Country =2

 id  name   population

│  1 │ Palau │       17900 │
│  2 │ Nauru │       11001 │
└────────────────────────

// Or not in
>> Country[name !in ["Palau", "Nauru"]]
      table Country =1

 id  name    population
```

(continues on next page)

(continued from previous page)

```
│  3 │ Tuvalu │      10201 │
└───────────────────────────
```

8.6 The SQL Escape-hatch

Preql does not, and cannot, implement every SQL function and feature.

There are too many dialects of SQL, and too few Preql programmers (for now).

Luckily, there is an escape hatch, through the `SQL()` function.

The first argument is the type of the result, and the second argument is a string of SQL code.

```
>> func do_sql_stuff(x) = SQL(string, "lower($x) || '!'")   // Runs in Sqlite
>> ["UP", "Up", "up"]{ do_sql_stuff(item) }
   table  =3

 do_sql_stuff

│ up!          │
│ up!          │
│ up!          │
└──────────────
```

We can also query entire tables:

```
 >> SQL(Country, "SELECT * FROM $Country WHERE name == \"Palau\"")
      table Country =1

 id  name   population

│  1 │ Palau │      17900 │
└────────────────────────
```

Notice that "Country" is used twice in different contexts: once as the return type, and once for querying its rows.

In fact, many of Preql's core functions are written using the `SQL()` function, for example `enum`:

```
func enum(tbl) {
    "Return the table with a new index column"
    // Uses SQL's window functions to calculate the index per each row
    // Remember, ellipsis (...) includes all available columns.
    return tbl{
      index: SQL(int, "row_number() over ()")
      ...
    }
}

// Add an index for each row in the table
>> enum(Country order {population})
             table  =3

 index  id  name    population
```

(continues on next page)

(continued from previous page)

```
|      0 |   3 | Tuvalu |         10201 |
|      1 |   2 | Nauru  |         11001 |
|      2 |   1 | Palau  |         17900 |
```

8.7 Notable Built-in functions

Here is a partial list of functions provided by Preql:

- `debug()` - call this from your code to drop into the interpreter. Inside, you can use `c()` or `continue()` to resume running.
- `import_csv(table, filename)` - import the contents of a csv file into an existing table
- `random()` - return a random number
- `now()` - return a `datetime` object for now
- `sample_fast(tbl, n)` - return a sample of `n` rows from table `tbl` (O(n), maximum of two queries). May introduce a minor bias (See `help(sample_fast)`).
- `bfs(edges, initial)` - performs a breadth-first search on a graph using SQL
- `count_distinct(field)` - count how many unique values are in the given field/column.

To see the full list, run the following in Preql: `names(__builtins__)[type like "function%"]`

8.8 Calling Preql from Python

Preql is not only a standalone tool, but also a Python library. It can be used as an alternative to ORM libraries such as SQLAlchemy.

It's as easy as:

```
>>> import preql
>>> p = preql.Preql()
```

You can also specify which database to work on, and other parameters.

Then, just start working by calling the object with Preql code:

```
# Use the result like in an ORM
>>> len(p('[1,2,3][item>=2]'))
2

# Turn the result to JSON (lists and dicts)
>>> p('[1,2]{type: "example", values: {v1: item, v2: item*2}}').to_json()
[{'type': 'example', 'values': {'v1': 1, 'v2': 2}}, {'type': 'example', 'values': {'v1
↪': 2, 'v2': 4}}]

# Run Preql code file
>>> p.load('some_file.pql')
```

You can also reach variables inside the Preql namespace using:

```
>>> p('a = [1,2,3]')
>>> sum(p.a)
6
>>> p.char_range('w', 'z')     # char_range() is a built-in function in Preql
['w', 'x', 'y', 'z']
```

8.8.1 Using Pandas

You can easily import/export tables between Preql and Pandas, by using Python as a middleman:

```
>>> from pandas import DataFrame
>>> import preql
>>> p = preql.Preql()
>>> f = DataFrame([[1,2,"a"], [4,5,"b"], [7,8,"c"]], columns=['x', 'y', 'z'])

>>> x = p.import_pandas(x=f)
>>> p.x                             # Returns a Preql table
[{'x': 1, 'y': 2, 'z': 'a', 'id': 1}, {'x': 4, 'y': 5, 'z': 'b', 'id': 2}, {'x': 7, 'y
↪': 8, 'z': 'c', 'id': 3}]

>>> p('x {y, z}').to_pandas()     # Returns a pandas table
   y  z
0  2  a
1  5  b
2  8  c

>>> p('x{... !id}').to_pandas() == f  # Same as it ever was
      x     y     z
0  True  True  True
1  True  True  True
2  True  True  True
```

CHAPTER 9

Code comparison: Preql, SQL and the rest

This document was written with the aid of Pandas' comparison with SQL.

Use the checkboxes to hide/show the code examples for each language.

9.1 Table Operations

9.1.1 Selecting columns

Column selection is done using the projection operator, `{}`.

```
tips{total_bill, tip, smoker, time}
```

The table name (`tips`) comes first, so that Preql can automatically suggest the field names.

In SQL, selection is done using the SELECT statement

```
SELECT total_bill, tip, smoker, time FROM tips;
```

```
tips[['total_bill', 'tip', 'smoker', 'time']]
```

9.1.2 Filtering rows

Row filtering is done using the filter operator, `[]`:

```
tips[size >= 5 or total_bill > 45]
```

```
SELECT * FROM tips WHERE size >= 5 OR total_bill > 45;
```

DataFrames can be filtered in multiple ways; Pandas suggest using boolean indexing:

```
tips[(tips['size'] >= 5) | (tips['total_bill'] > 45)]
```

```
from sqlalchemy import or_
session.query(Tips).filter(or_(Tips.size >= 5, Tips.total_bill > 45))
```

9.1.3 Group by / Aggregation

In this example, we calculate how the amount of tips differs by day of the week.

Preql extends the projection operator to allow aggregation using the => construct:

```
tips{day => avg(tip), count()}
```

Conceptually, everything on the left of => are the keys, and on the right are the aggregated values.

```
SELECT day, AVG(tip), COUNT(*) FROM tips GROUP BY day;
```

```
tips.groupby('day').agg({'tip': np.mean, 'day': np.size})
```

```
from sqlalchemy import func
session.query(Tips.day, func.avg(Tips.tip), func.count(Tips.id)).group_by(Tips.day).
↪all()
```

9.1.4 Concat, Union

In this example, we will concatenate and union two tables together.

```
table1 + table2         // concat
```

```
table1 | table2         // union
```

```
SELECT * FROM table1 UNION ALL SELECT * FROM table2;   -- concat
```

```
SELECT * FROM table1 UNION SELECT * FROM table2;       -- union
```

```
pd.concat([table1, table2])                        # concat
```

```
pd.concat([table1, table2]).drop_duplicates()      # union
```

```
union_all(session.query(table1), session.query(table2))       # concat
```

```
union(session.query(table1), session.query(table2))           # union
```

9.1.5 Top n rows with offset (limit)

```
tips[5..15]
// OR
tips[5..][..10]
```

```
SELECT * FROM tips ORDER BY tip DESC LIMIT 10 OFFSET 5;
```

```
tips.nlargest(10 + 5).tail(10)
```

9.1.6 Join

Join is essentially an operation that matches rows between two tables, based on common attributes.

```
join(a: table1.key1, b: table2.key2)
```

The result is a table with two columns, `a` and `b`, which are structs that each contain the columns of their respective table.

If we have pre-defined a "default join" between tables, we can shorten it to:

```
join(a: table1, b: table2)
```

Preql also offers the functions `leftjoin()`, `outerjoin()`, and `joinall()`.

```
SELECT * FROM table1 INNER JOIN table2 ON table1.key1 = table2.key2;
```

```
pd.merge(df1, df2, on='key')
```

(it gets complicated if the key isn't with the same name)

```
session.query(Table1).join(Tables2).filter(Table1.key1 == Table2.key2)
```

9.1.7 Insert row

Insert a row to the table, and specifying the columns by name.

```
new Country(name: "Spain", language: "Spanish")
```

```
INSERT INTO Country (name, language) VALUES ("Spain", "Spanish")
```

```
countries = countries.append({'name':'Spain', 'language': 'Spanish'}, ignore_
↪index=True)
```

```
session.add(Country(name='Spain', language='Spanish'))
```

9.1.8 Update rows

```
tips[tip < 2] update {tip: tip*2}
```

Preql puts the `update` keyword after the selection, so that when working interactively, you can first see which rows you're about to update.

```
UPDATE tips SET tip = tip*2 WHERE tip < 2;
```

```
tips.loc[tips['tip'] < 2, 'tip'] *= 2
```

(takes a different form for complex operations)

9.2 Gotchas

9.2.1 Null checks

Comparisons to `null` behave like in Python.

```
tips[col2==null]
```

Preql also has a value called `unknown`, which behaves like SQL's `NULL`.

Simple comparison to `NULL` using =, will always return `NULL`. For comparing to `NULL`, you must use the `IS` operator (the operator name changes between dialects).

```
SELECT * FROM tips WHERE col2 IS NULL;
```

```
tips[tips['col2'].isna()]
```

9.3 Programming

9.3.1 Defining a function, and calling it from the query

```
func add_one(x: int) = x + 1

my_table{ add_one(my_column) }
```

(Type annotations validate the values at compile-time)

(Postgres dialect)

```
CREATE FUNCTION add_one(x int)
RETURNS int
AS
$$
 SELECT x + 1
$$
LANGUAGE SQL IMMUTABLE STRICT;

SELECT add_one(my_column) FROM my_table;
```

```
def add_one(x: int):
    return x + 1

my_table['my_column'].apply(add_one)
```

Impossible?

9.3.2 Counting a table from Python

This example demonstrates Preql's Python API.

All examples set `row_count` to an integer value.

(assumes `p` is a preql instance)

```
row_count = len(p.my_table)
```

Or:

```
row_count = p('count(my_table)')
```

```
cur = conn.execute('SELECT COUNT() FROM my_table')
row_count = cur.fetchall()[0][0]
```

```
row_count = len(my_table.index)
```

```
row_count = session.query(my_table).count()
```

CHAPTER 10

Getting Started

10.1 Install

1. Ensure you have Python 3.6, or above, installed on your system.
2. Ensure you have pip for Python (you probably already do).
3. Run the following command:

```
pip install -U preql
```

10.2 Run the interpreter in the console (REPL)

To start the interpreter, run the following in your shell:

```
preql
```

Preql will use Sqlite's memory database by default.

To see the running options, type:

```
preql --help
```

10.2.1 Explore an existing database

When you start the interpreter, you can specify which database to connect to, using a URL.

```
# Postgresql
preql postgres://user:pass@host/dbname

# MySQL
```

(continues on next page)

(continued from previous page)

```
preql mysql://user:pass@host/dbname

# Sqlite (use existing or create new)
preql sqlite://path/to/file
```

When already inside the Preql interactive prompt, a Jupyter Notebook, or a running script, use the `connect()` method:

```
connect("sqlite://path/to/file")
```

Use introspective methods to see a list of the tables, and of the available functions:

```
// Get a list of all tables in database
>> tables()

// Get help regarding how to use Preql
>> help()

// For example:
>> help(connect)
func connect(uri, load_all_tables, auto_create) = ...

    Connect to a new database, specified by the uri
    ...
```

10.3 Run in a Jupyter Notebook

1. Install the Preql kernel into jupyter:

```
preql --install-jupyter
```

1. Run Jupyter Notebook as usual:

```
jupyter notebook
```

1. create a new notebook with the `Preql` kernel, or open an existing one.

Inside the notebook, use the `connect()` function to connect to a database.

For an example, view the following Jupyter notebook: Tutorial: Exploring a database with Preql

10.4 Use as a Python library

```
from preql import Preql
p1 = Preql()                             # Use memory database
p2 = Preql("sqlite://path/to/file")      # Use existing or new file

assert p1('sum([1..10])') == 45
```

10.5 Run as a REST / GraphQL server

Coming soon!

10.6 Further reading

- *Learn the language*
- *Read the tutorial*

CHAPTER 11

Why not SQL/ORM/Pandas ?

11.1 SQL

SQL first appeared in 1974, and aimed to provide a database interface that was based on natural language. It was clever and innovative at the time of its conception, but today we can look back on its design and see many fundamental mistakes.

Among its many faults, SQL is excessively verbose, is bad at catching and reporting errors, has no first-class or high-order functions, is awkward for interactive work, and it has a fragmented ecosystem and many different and incompatible dialects.

Some attempts have been made to address these issues, mainly in the form of ORMs.

11.1.1 Good parts of SQL

- Relational Algebra
- Declarative queries
- Many mature, well-tested implementations
- Intended for interactive work

11.1.2 Bad parts of SQL

- Lack of first-class functions
- Hard to re-use code
- Bad error-handling (if any)
- Long-winded and clumsy syntax
- Code lives on the server, so there is no version control (such as git)
- Interactive clients leave a lot to be desired

(there are plenty more on both sides of the scales)

Preql adopts the good parts of SQL, and tries to solve the bad parts.

11.2 ORMs

ORMs (object-relational mapping), are frameworks that let their users interact with the database using constructs that are native to the host programming language. Those constructs are then compiled to SQL, and executed in the database.

ORMs are usually more concise and more composable than SQL. However, they are themselves limited by their host languages, which were never designed for relational data processing. For the most part, they have awkward syntax, and they only support simple constructs and queries, and simplistic compostion.

11.3 Pandas

Given the failings of SQL and ORMs, it's no wonder that many programmers and data analysts choose to disregard relational databases altogether, and use completely new approaches.

Pandas is one of those new approaches. Implemented entirely on top of Python and Numpy, it has gained a lot of popularity in recent years due to its accessibility and relative simplicity. It also has a wide range of features that were designed specifically for data scientists.

Unfortunately, it comes with its own set of faults. Pandas is slow (despite recent efforts to accelerate it), it has awkward syntax, and it isn't well suited for working with relational, structured or linked data.

See also: Code comparison

CHAPTER 12

Language Reference

(This document is incomplete, and needs more work)

Preql's syntax is a mix between Go and Javascript.

- Comments start with //

12.1 Literals

12.1.1 Null

Null values are specified with `null`. Null is only ever equal to itself:

```
 >> null == null
true
```

12.1.2 Booleans

The two boolean values are `true` and `false`, and both are of type `bool`.

```
 >> type(true)
bool
 >> type(false)
bool
```

12.1.3 Numbers

Numbers are written as integers or floats.

Standard operators apply: `+`, `-`, `*`, `/`, `%`, `**`, `==`, `!=`, `<`, `>`, `<=`, `>=`

```
>> type(10)
int
>> type(3.14)
float
```

Operations between ints and floats result in a float:

```
>> type(10 + 3.14)
float
```

Division always returns a float. For “floordiv”, use the `/~` operator:

```
>> 10 / 3
3.3333333333333335
>> 10 /~ 3
3
```

12.1.4 Strings

Standard operators apply: `+`, `*`, `==`, `!=`

Strings, like in Python, take one of the following forms:

- `'a'`
- `"a"`
- `'''a'''`
- `"""a"""`

```
>> type("a")
string
```

Triple-quoted strings capture newlines, while single-quoted strings do not.

Strings support the `like` operator, or `~`, similar to SQL’s like operator:

```
>> "hello" ~ "h%"
true
```

Strings support the slicing operators `[]`:

```
>> "preql"[3..]
"ql"
```

12.1.5 Structs

Structs can be created on the fly, using the `{}` syntax:

Structs are essentially dictionaries (or maps), in which the keys are always of the type string.

```
>> x = {a:1, b:2}
{a: 1, b: 2}
>> x.a
1
```

(continues on next page)

(continued from previous page)

```
>> type(x)
struct[a: int, b: int]
```

12.1.6 Lists

Lists can be specified using the `[item1, item2, ...]` syntax. They are equivalent to a table with a single `item` column.

Lists support all tables operations.

```
>> ["a", "b", "c"]
table  =3
───────
│ item  │
├───────
│ a     │
│ b     │
│ c     │
└───────
```

12.1.7 Ranges

Ranges can be specified using the `[start..end]` syntax. They are equivalent to a list of numbers.

```
>> type([1..10])
list[int]
```

12.1.8 Functions

- Functions are defined with `func`, like in Go

```
// normal syntax
func abs(x) {
    "docstring"
    if (x < 0) {
        return -x
    }
    return x
}

// short-hand syntax
func add1(x) = x + 1
    "docstring"
```

12.2 Keywords

12.2.1 Operators

CHAPTER 13

Preql Types

any

A meta-type that can match any type.

Subtypes unknown, type, object,

Examples

```
>> isa(my_obj, any)          // always returns true
true
>> isa(my_type, any)         // always returns true
true
```

union

A meta-type that means 'either one of the given types'

Example

```
>> int <= union[int, string]
true
>> union[int, string] <= int
false
>> union[int, string] <= union[string, int]
true
```

type

The type of types

Supertypes any

Examples

```
>> type(int) == type(string)
true
>> int <= type(int)                  # int isn't a subtype of `type`
false
```

(continues on next page)

(continued from previous page)

```
>> isa(int, type(int))            # int is an instance of `type`
true
```

object
The base object type

Supertypes any

Subtypes nulltype, primitive, container, aggregate_result[any], function, property, module, signal,

nulltype
The type of the singleton *null*. Represents SQL *NULL*, but behaves like Python's *None*,

Supertypes object

Examples

```
>> null == null
true
>> null + 1
[bold]TypeError[/bold]: Operator '+' not implemented for nulltype and␣
↪int
```

primitive
The base type for all primitives

Supertypes object

Subtypes text, number, bool, timestamp, datetime, date, time, t_id[table], t_relation[any],

text
A text type (behaves the same as *string*)

Supertypes primitive

Subtypes _rich, string,

string
A string type (behaves the same as *text*)

Supertypes text

number
The base type for all numbers

Supertypes primitive

Subtypes int, float, decimal,

int
An integer number

Supertypes number

float
A floating-point number

Supertypes number

bool
A boolean, which can be either *true* or *false*

Supertypes primitive

timestamp
A timestamp type (unix epoch)

Supertypes primitive

Methods

datetime
A datetime type (date+time combined)

Supertypes primitive

container
The base type of containers. A container holds other objects inside it.

Supertypes object

Subtypes struct, table, aggregated[any], projected[any], json[any],

struct
A structure type

Supertypes container

Subtypes row,

row
A row in a table. (essentially a named-tuple)

Supertypes struct

table

A table type. Tables support the following operations - - Projection (or: map), using the *{}* operator - Selection (or: filter), using the *[]* operator - Slice (or: indexing), using the *[..]* operator - Order (or: sorting), using the *order{}* operator - Update, using the *update{}* operator - Delete, using the *delete[]* operator - + for concat, & for intersect, | for union

Supertypes container

Subtypes list[item: any], set[item: any],

Methods

add_index (*column_name*, *unique*)
Add an index to the table, to optimize filtering operations. A method of the *table* type.

Parameters

- **column_name** (*union[string, list[item: string]]*) – The name of the column to add index
- **unique** (*bool*) – If true, every value in the column is expected to be unique (default=false)

Note Future versions of this function will accept several columns.

Example

```
>> table x = [1,2,3]{item}
>> x.add_index("item")
```

list[item: any]
A list type

Supertypes table

set[item: any]
A set type, in which all elements are unique

Supertypes table

t_id[table]
The type of a table id

Supertypes primitive

t_relation[any]
The type of a table relation

Supertypes primitive

aggregated[any]
A meta-type to signify aggregated operations, i.e. operations inside a grouping

Supertypes container

Example

```
>> x = [1]
>> one one x{ => repr(type(item))}
"aggregated[item: int]"
```

projected[any]
A meta-type to signify projected operations, i.e. operations inside a projection.

Supertypes container

Example

```
>> x = [1]
>> one one x{ repr(type(item)) }
"projected[item: int]"
```

json[any]
A json type

Supertypes container

Subtypes json_array,

json_array
A json array type. Created by aggregation.

Supertypes json[any]

function
A meta-type for all functions

Supertypes object

module
A meta-type for all modules

Supertypes object

signal
A meta-type for all signals (i.e. exceptions)

Supertypes object

Subtypes Exception,

CHAPTER 14

Preql Modules

14.1 __builtins__

PY (*code_expr*, *code_setup*)

Evaluate the given Python expression and convert the result to a Preql object

Parameters

- **code_expr** (*string*) – The Python expression to evaluate
- **code_setup** (*string?*) – Setup code to prepare for the evaluation (default=null)

Note This function is still experemental, and should be used with caution.

Example

```
>> PY("sys.version", "import sys")
"3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 23:03:10)"
```

SQL (*result_type*, *sql_code*)

Create an object with the given SQL evaluation code, and given result type. The object will only be evaluated when required by the program flow. Using $var_name in the code will embed it in the query. Both primitives and tables are supported. A special *$self* variable allows to perform recursion, if supported by the dialect.

Parameters

- **result_type** (*union[table, type]*) – The expected type of the result of the SQL query
- **sql_code** (*string*) – The SQL code to be evaluated

Example

```
>> ["a", "b"]{item: SQL(string, "$item || '!'")}
table  =2

 item
```

(continues on next page)

(continued from previous page)

```
│ a!     │
│ b!     │
└────────
>> x = ["a", "b", "c"]
>> SQL(type(x), "SELECT item || '!' FROM $x")
table  =3

 item

│ a!     │
│ b!     │
│ c!     │
└────────
```

approx_product (*col*)
: Returns the approximate product of an aggregated column. It does so using logarithmic math. See *product* for an accurate version of this function.

  Parameters **col** (*aggregated[item: number]*) –

cast (*obj*, *target_type*)
: Attempt to cast an object to a specified type The resulting object will be of type *target_type*, or a *TypeError* exception will be thrown.

  Parameters
  - **obj** (*any*) – The object to cast
  - **target_type** (*type*) – The type to cast to

char (*n*)
: Returns the character with the given ASCII code

  Parameters **n** (*int*) –

char_ord (*n*)
: Returns the ascii code of the given character

  Parameters **n** (*string*) –

char_range (*start*, *end*)
: Produce a list of all characters from 'start' to 'stop'

  Parameters
  - **start** (*string*) –
  - **end** (*string*) –

  Example

```
>> char_range('a', 'z')
```

columns (*obj*)
: Returns a dictionary *{column_name: column_type}* for the given table

  Parameters **obj** (*container*) –

  Example

```
>> columns([0])
{item: int}
```

commit ()
Commit the current transaction This is necessary for changes to the tables to become persistent.

connect (*uri*, *load_all_tables*, *auto_create*)
Connect to a new database, specified by the uri

Parameters

- **uri** (*string*) – A string specifying which database to connect to (e.g. “sqlite:///test.db”)
- **load_all_tables** (*bool*) – If true, loads all the tables in the database into the global namespace. (default=false)
- **auto_create** (*bool*) – If true, creates the database if it doesn’t already exist (Sqlite only) (default=false)

Example

```
>> connect("sqlite://:memory:")     // Connect to a database in memory
```

count (*obj*)
Count how many rows are in the given table, or in the projected column. If no argument is given, count all the rows in the current projection.

Parameters **obj** (*container?*) –

Examples

```
>> count([0..10])
10
>> [0..10]{ => count() }
table  =1

 count

    10 |

>> [0..10]{ => count(item) }
table  =1

 count

    10 |
```

count_false (*field*)
Count how many values in the field are false or zero

Parameters **field** (*aggregated[any]*) –

Example

```
>> [0,1,2,0,3]{ => count_false(item) }
table  =1

 count_false

```

(continues on next page)

(continued from previous page)

```
│          2 │
└────────────┘
```

See Also

- *count_true*

count_true (*field*)
Count how many values in the field are true (non-zero)

Parameters **field** (*aggregated[any]*) –

Example

```
>> [0,1,2,0,3]{ => count_true(item) }
table  =1

 count_true

│          3 │
└────────────┘
```

See Also

- *count_false*

debug ()
Breaks the execution of the interpreter, and enters into a debug session using the REPL environment. Use *c()* to continue the execution.

dict (*...x*)
Constructs a dictionary

dir (*obj*)
List all names in the namespace of the given object. If no object is given, lists the names in the current namespace.

Parameters **obj** (*any*) –

distinct (*t*)
Removes identical rows from the given table

Parameters **t** (*table*) –

Example

```
>> distinct(["a","b","b","c"])
table  =3

 item

│ a      │
│ b      │
│ c      │
└────────┘
```

env_vars ()
Returns a table of all the environment variables. The resulting table has two columns: name, and value.

exit (*value*)
Exit the current interpreter instance. Can be used from running code, or the REPL. If the current interpreter is nested within another Preql interpreter (e.g. by using debug()), exit() will return to the parent interpreter.

Parameters **value** (*any?*) –

first (*obj*)
Returns the first member of a column or a list

Parameters **obj** (*union[table, aggregated[any]]*) –

Examples

```
>> first([1,2,3])
1
>> [1,2,3]{ => first(item) }
table  =1

 first

│      1 │
└────────
```

first_or_null (*obj*)
Returns the first member of a column or a list, or null if it's empty See Also *first()*.

Parameters **obj** (*union[table, projected[any]]*) –

fmt (*s*)
Format the given string using interpolation on variables marked as *$var*

Parameters **s** (*string*) –

Example

```
>> ["a", "b", "c"]{item: fmt("$item!")}
table  =3

 item

│ a!     │
│ b!     │
│ c!     │
└────────
```

force_eval (*expr*)
Forces the evaluation of the given expression. Executes any db queries necessary.

Parameters **expr** (*object*) –

get_db_type ()
Returns a string representing the type of the active database.

Example

```
>> get_db_type()
"sqlite"
```

help (*inst*)
Provides a brief summary for the given object

Parameters **inst** (*any*) –

import_csv (*table*, *filename*, *header*)
Import a csv file into an existing table

Parameters

- **table** (*table*) – A table into which to add the rows.
- **filename** (*string*) – A path to the csv file
- **header** (*bool*) – If true, skips the first line (default=false)

import_json (*table_name*, *uri*)
Imports a json file into a new table. Returns the newly created table.

Parameters

- **table_name** (*string*) – The name of the table to create
- **uri** (*string*) – A path or URI to the JSON file

Note This function requires the *pandas* Python package.

import_table (*name*, *columns*)
Import an existing table from the database, and fill in the types automatically.

Parameters

- **name** (*string*) – The name of the table to import
- **columns** (*list?[item: string]*) – If this argument is provided, only these columns will be imported. (default=null)

Example

```
>> import_table("my_sql_table", ["some_column", "another_column])
```

inspect_sql (*obj*)
Returns the SQL code that would be executed to evaluate the given object

Parameters **obj** (*object*) –

is_empty (*tbl*)
Efficiently tests whether the table expression *tbl* is empty or not

Parameters **tbl** –

isa (*obj*, *type*)
Checks if the give object is an instance of the given type

Parameters

- **obj** (*any*) –
- **type** (*type*) –

Examples

```
>> isa(1, int)
true
>> isa(1, string)
false
>> isa(1.2, number)
true
>> isa([1], table)
true
```

issubclass (*a*, *b*)

Checks if type 'a' is a subclass of type 'b'

Parameters

- **a** (*type*) –
- **b** (*type*) –

Examples

```
>> issubclass(int, number)
true
>> issubclass(int, table)
false
>> issubclass(list, table)
true
```

join (*$on*, *...tables*)

Inner-join any number of tables. Each argument is expected to be one of - (1) A column to join on. Columns are attached to specific tables. or (2) A table to join on. The column will be chosen automatically, if there is no ambiguity. Connections are made according to the relationships in the declaration of the table.

Parameters

- **$on** – Optional special keyword argument for specifying join condition. When specified, auto-join will be skipped. (default=null)
- **tables** – Provided as keyword arguments, in the form of <name>: <table>. Each keyword argument must be either a column, or a table.

Returns A new table, where each column is a struct representing one of the joined tables.

Examples

```
>> join(a: [0].item, b: [0].item)
table join46 =1

 a             b

│ {'item': 0} │ {'item': 0} │
└─────────────────────────────

>> join(a: [1..5].item, b: [3..8].item) {...a}
table  =2

  item

│      3 │
│      4 │
└────────

>> leftjoin(a: [1,3], b: [1,2], $on: a.item > b.item)
table join78 =3

 a             b

│ {'item': 1} │ {'item': None} │
│ {'item': 3} │ {'item': 1}    │
│ {'item': 3} │ {'item': 2}    │
└────────────────────────────────

>> join(c: Country, l: Language) {...c, language: l.name}
```

joinall(*$on*, *...tables*)
Cartesian product of any number of tables See *join*

Parameters **$on** –

Example

```
>> joinall(a: [0,1], b: ["a", "b"])
table joinall14 =4

 a             b

 │ {'item': 0} │ {'item': 'a'} │
 │ {'item': 0} │ {'item': 'b'} │
 │ {'item': 1} │ {'item': 'a'} │
 │ {'item': 1} │ {'item': 'b'} │
 └─────────────────────────────┘
```

leftjoin(*$on*, *...tables*)
Left-join any number of tables See *join*

Parameters **$on** –

length(*s*)
Returns the length of the string For tables or lists, use *count()*

Parameters **s** (*string*) –

limit(*tbl*, *n*)
Returns the first 'n' rows in the table.

Parameters

- **tbl** (*table*) –
- **n** (*int*) –

limit_offset(*tbl*, *lim*, *offset*)
Returns the first 'n' rows in the table at the given offset.

Parameters

- **tbl** (*table*) –
- **lim** (*int*) –
- **offset** (*int*) –

list_median(*x*)
Find the median of a list Cannot be used inside a projection.

Parameters **x** (*list[item: any]*) –

lower(*s*)
Return a copy of the string converted to lowercase.

Parameters **s** (*string*) –

map_range(*tbl*, *start*, *end*)
For each row in the table, assigns numbers out of a range, and produces *(end-start)* new rows instead, each attached to a number. If *start* or *end* are functions, the index is the result of the function, per row.

Parameters

- **tbl** (*table*) – Table to map the range onto

- **start** (*union[int, function]*) – The starting index, or a function producing the starting index
- **end** (*union[int, function]*) – The ending index, or a function producing the ending index

Examples

```
>> map_range(["a", "b"], 0, 3)
table  =6

 i  item

│ 0 │ a    │
│ 1 │ a    │
│ 2 │ a    │
│ 0 │ b    │
│ 1 │ b    │
│ 2 │ b    │

>> map_range(["a", "ab"], 1, length)
table  =3

 i  item

│ 1 │ a    │
│ 1 │ ab   │
│ 2 │ ab   │
```

max (*col*)
Finds the maximum of a column or a list See Also *sum*.

Parameters col (*union[table[item: number], aggregated[item: number]]*) –

mean (*col*)
Returns the mean average of a column or a list See Also *sum*.

Parameters col (*union[table[item: number], aggregated[item: number]]*) –

min (*col*)
Finds the minimum of a column or a list See Also *sum*.

Parameters col (*union[table[item: number], aggregated[item: number]]*) –

names (*obj*)
List all names in the namespace of the given object. If no object is given, lists the names in the current namespace.

Parameters obj (*any*) –

now ()
Returns the current timestamp

outerjoin (*$on, ...tables*)
Outer-join any number of tables See *join*

Parameters $on –

page (*table*, *index*, *page_size*)
Pagination utility function for tables

Parameters

- **table** –
- **index** –
- **page_size** –

product (*col*)
Returns the product of a column or a list See Also *sum*.

Parameters **col** (*union[table[item: number], aggregated[item: number]]*) –

Note This function is only available in sqlite3 by default. To make it available in postgres, users must call the *install_polyfills()* function. For databases that don't support product, see *approx_product()*.

random ()
Returns a random float number between 0 to 1

remove_table (*table_name*)
Remove table from database (drop table)

Parameters **table_name** –

remove_table_if_exists (*table_name*)
Remove table from database (drop table). Ignore if it doesn't exist.

Parameters **table_name** –

repeat (*s*, *num*)
Repeats the string *num* times.

Parameters

- **s** (*string*) –
- **num** (*int*) –

Example

```
>> _repeat("ha", 3)
"hahaha"
```

repr (*obj*)
Returns the representation text of the given object

Parameters **obj** (*any*) –

rollback ()
Rollback the current transaction This reverts the data in all the tables to the last commit. Local variables will remain unaffected.

round (*n*, *precision*)
Returns a rounded float at the given precision (i.e. at the given digit index)

Parameters

- **n** (*number*) –
- **precision** (*int*) –

Example

```
>> round(3.14)
3.0
>> round(3.14, 1)
3.1
```

sample_fast (*tbl*, *n*, *bias*)
Returns a random sample of n rows from the table in one query (or at worst two queries)

Parameters

- **tbl** (*table*) – The table to sample from
- **n** (*int*) – The number of items to sample
- **bias** (*number*) – Add bias (reduce randomness) to gain performance. Higher values of 'bias' increase the chance of success in a single query, but may introduce a higher bias in the randomness of the chosen rows, especially in sorted tables. (default=0.05)

sample_ratio_fast (*tbl*, *ratio*)
Returns a random sample of rows from the table, at the approximate amount of (ratio*count(tbl)).

Parameters

- **tbl** –
- **ratio** –

serve_rest (*endpoints*, *port*)
Start a starlette server (HTTP) that exposes the current namespace as REST API

Parameters

- **endpoints** (*struct*) – A struct of type (string => function), mapping names to the functions.
- **port** (*int*) – A port from which to serve the API (default=8080)

Note Requires the *starlette* package for Python. Run *pip install starlette*.

Example

```
>> func index() = "Hello World!"
>> serve_rest({index: index})
INFO     Started server process [85728]
INFO     Waiting for application startup.
INFO     Application startup complete.
INFO     Uvicorn running on http://127.0.0.1:8080 (Press CTRL+C to␣
↪quit)
```

stddev (*col*)
Finds the standard deviation of a column or a list See Also *sum*.

Parameters **col** (*union[table[item: number], aggregated[item: number]]*) –

str_contains (*substr*, *s*)
Tests whether string *substr* is contained in *s*

Parameters

- **substr** (*string*) –
- **s** (*string*) –

Example

```
>> str_contains("i", "tim")
true
>> str_contains("i", "team")
false
```

str_index (*substr*, *s*)

Finds in which index does *substr* appear in *s*.

Parameters

- **substr** (*string*) – The substring to find
- **s** (*string*) – The string to search in

Returns A 0-based index (int) if found the substring, or -1 if not found.

Example

```
>> str_index("re", "preql")
1
>> str_index("x", "preql")
-1
```

str_notcontains (*substr*, *s*)

Tests whether string *substr* is not contained in *s* Equivalent to *not str_contains(substr, s)*.

Parameters

- **substr** (*string*) –
- **s** (*string*) –

sum (*col*)

Sums up a column or a list.

Parameters **col** (*union[table[item: number], aggregated[item: number]]*) –

Examples

```
>> sum([1,2,3])
6
>> [1,2,3]{ => sum(item) }
table  =1

   sum

 │     6 │
 └───────
```

table_concat (*t1*, *t2*)

Concatenate two tables (union all). Used for *t1 + t2*

Parameters

- **t1** (*table*) –
- **t2** (*table*) –

table_intersect (*t1*, *t2*)

Intersect two tables. Used for *t1 & t2*

Parameters

- **t1** (*table*) –
- **t2** (*table*) –

table_subtract (*t1*, *t2*)
Substract two tables (except). Used for *t1 - t2*

Parameters

- **t1** (*table*) –
- **t2** (*table*) –

table_union (*t1*, *t2*)
Union two tables. Used for *t1 | t2*

Parameters

- **t1** (*table*) –
- **t2** (*table*) –

tables ()
Returns a table of all the persistent tables in the database. The resulting table has two columns: name, and type.

temptable (*expr*, *const*)
Generate a temporary table with the contents of the given table It will remain available until the database session ends, unless manually removed.

Parameters

- **expr** (*table*) – the table expression to create the table from
- **const** (*bool?*) – whether the resulting table may be changed or not. (default=null)

Note A non-const table creates its own *id* field. Trying to copy an existing id field into it will cause a collision

type (*obj*)
Returns the type of the given object

Parameters **obj** (*any*) –

Example

```
>> type(1)
int
>> type([1])
list[item: int]
>> type(int)
type
```

upper (*s*)
Return a copy of the string converted to uppercase.

Parameters **s** (*string*) –

zipjoin (*a*, *b*)
Joins two tables on their row index. Column names are always *a* and *b*. Result is as long as the shortest table. Similar to Python's *zip()* function.

Parameters

- **a** (*table*) –

- **b** (*table*) –

Example

```
>> zipjoin(["a", "b"], [1, 2])
table  =2

 a              b

│ {'item': 'a'} │ {'item': 1} │
│ {'item': 'b'} │ {'item': 2} │
└────────────────────────────┘
```

zipjoin_left (*a*, *b*)

Similar to *zipjoin*, but the result is as long as the first parameter. Missing rows will be assigned *null*.

Parameters

- **a** (*table*) –
- **b** (*table*) –

Example

```
>> zipjoin_left(["a", "b"], [1])
table  =2

 a              b

│ {'item': 'a'} │ {'item': 1}    │
│ {'item': 'b'} │ {'item': null} │
└───────────────────────────────┘
```

zipjoin_longest (*a*, *b*)

Similar to *zipjoin*, but the result is as long as the longest table. Missing rows will be assigned *null*.

Parameters

- **a** (*table*) –
- **b** (*table*) –

14.2 graph

bfs (*edges*, *initial*)

Performs a breadth-first search on a graph.

Parameters

- **edges** (*table*) – a table of type *{src: int, dst: int}*, defining the edges of the graph
- **initial** (*table*) – list[int], specifies from which nodes to start

walk_tree (*edges*, *initial*, *max_rank*)

Walks a tree and keeps track of the rank. Doesn't test for uniqueness. Nodes may be visited more than once. Cycles will repeat until max_rank.

Parameters

- **edges** (*table*) – a table of type {src: int, dst: int}, defining the edges of the graph

- **initial** (*table*) – list[int], specifies from which nodes to start
- **max_rank** (*int*) – integer limiting how far to search

CHAPTER 15

Python API Reference

15.1 Preql

class preql.**Preql**(*db_uri: str = 'sqlite://:memory:'*, *print_sql: bool = False*, *auto_create: bool = False*, *autocommit: bool = False*)

Provides an API to run Preql code from Python

Example

```
>>> import preql
>>> p = preql.Preql()
>>> p('[1, 2]{item+1}')
[2, 3]
```

__init__(*db_uri: str = 'sqlite://:memory:'*, *print_sql: bool = False*, *auto_create: bool = False*, *autocommit: bool = False*)

Initialize a new Preql instance

Parameters

- **db_uri** (`str, optional`) – URI of database. Defaults to using a non-persistent memory database.
- **print_sql** (`bool, optional`) – Whether or not to print every SQL query that is executed (default defined in settings)

load(*filename*, *rel_to=None*)

Load a Preql script

Parameters

- **filename** (`str`) – Name of script to run
- **rel_to** (`Optional[str]`) – Path to which `filename` is relative.

start_repl(**args*)
Run the interactive prompt

import_pandas(***dfs*)
Import pandas.DataFrame instances into SQL tables

Example

```
>>> pql.import_pandas(a=df_a, b=df_b)
```

class preql.api.**TablePromise**(*interp*, *inst*)
Returned by Preql whenever the result is a table

Fetching values creates queries to database engine

to_json()
Returns table as a list of rows, i.e. `[{col1: value, col2: value, ...}, ...]`

to_pandas()
Returns table as a Pandas dataframe (requires pandas installed)

__eq__(*other*)
Compare the table to a JSON representation of it as list of objects

Essentially: `return self.to_json() == other`

__len__()
Run a count query on table

__getitem__(*index*)
Run a slice query on table

Preql (*pronounced: Prequel*) is an interpreted relational query language, that compiles to SQL.

It is designed for use by data engineers, analysts and data scientists.

It features modern syntax and semantics, Python integration, inline SQL, and an interactive enviroment.

It supports PostgreSQL, MySQL, SQLite, BigQuery (WIP), and soon more.

CHAPTER 16

Resources

- *Introduction*
- *Getting Started*
- *Features*
- *Performance*
- *Roadmap*
- *FAQ*
- Tutorials
  - *Tutorial for the basics of Preql*
  - *Code comparison: Preql, SQL and the rest*
  - Jupyter tutorial
- Propaganda
  - *Why not SQL/ORM/Pandas ?*
- Reference
  - *Language Reference*
  - *Preql Types*
  - *Preql Modules*
  - *Python API Reference*

Index

Symbols

__eq__() (*preql.api.TablePromise method*), 68
__getitem__() (*preql.api.TablePromise method*), 68
__init__() (*preql.Preql method*), 67
__len__() (*preql.api.TablePromise method*), 68

A

add_index(), 49
approx_product() (*built-in function*), 52

B

bfs() (*built-in function*), 64

C

cast() (*built-in function*), 52
char() (*built-in function*), 52
char_ord() (*built-in function*), 52
char_range() (*built-in function*), 52
columns() (*built-in function*), 52
commit() (*built-in function*), 53
connect() (*built-in function*), 53
count() (*built-in function*), 53
count_false() (*built-in function*), 53
count_true() (*built-in function*), 54

D

debug() (*built-in function*), 54
dict() (*built-in function*), 54
dir() (*built-in function*), 54
distinct() (*built-in function*), 54

E

env_vars() (*built-in function*), 54
exit() (*built-in function*), 54

F

first() (*built-in function*), 55
first_or_null() (*built-in function*), 55
fmt() (*built-in function*), 55
force_eval() (*built-in function*), 55

G

get_db_type() (*built-in function*), 55

H

help() (*built-in function*), 55

I

import_csv() (*built-in function*), 55
import_json() (*built-in function*), 56
import_pandas() (*preql.Preql method*), 68
import_table() (*built-in function*), 56
inspect_sql() (*built-in function*), 56
is_empty() (*built-in function*), 56
isa() (*built-in function*), 56
issubclass() (*built-in function*), 56

J

join() (*built-in function*), 57
joinall() (*built-in function*), 57

L

leftjoin() (*built-in function*), 58
length() (*built-in function*), 58
limit() (*built-in function*), 58
limit_offset() (*built-in function*), 58
list_median() (*built-in function*), 58
load() (*preql.Preql method*), 67
lower() (*built-in function*), 58

M

map_range() (*built-in function*), 58
max() (*built-in function*), 59
mean() (*built-in function*), 59
min() (*built-in function*), 59

N

names() (*built-in function*), 59

now() (*built-in function*), 59

O

outerjoin() (*built-in function*), 59

P

page() (*built-in function*), 59
Preql (*class in preql*), 67
product() (*built-in function*), 60
PY() (*built-in function*), 51

R

random() (*built-in function*), 60
remove_table() (*built-in function*), 60
remove_table_if_exists() (*built-in function*), 60
repeat() (*built-in function*), 60
repr() (*built-in function*), 60
rollback() (*built-in function*), 60
round() (*built-in function*), 60

S

sample_fast() (*built-in function*), 61
sample_ratio_fast() (*built-in function*), 61
serve_rest() (*built-in function*), 61
SQL() (*built-in function*), 51
start_repl() (*preql.Preql method*), 67
stddev() (*built-in function*), 61
str_contains() (*built-in function*), 61
str_index() (*built-in function*), 62
str_notcontains() (*built-in function*), 62
sum() (*built-in function*), 62

T

table_concat() (*built-in function*), 62
table_intersect() (*built-in function*), 62
table_subtract() (*built-in function*), 63
table_union() (*built-in function*), 63
TablePromise (*class in preql.api*), 68
tables() (*built-in function*), 63
temptable() (*built-in function*), 63
to_json() (*preql.api.TablePromise method*), 68
to_pandas() (*preql.api.TablePromise method*), 68
type() (*built-in function*), 63

U

upper() (*built-in function*), 63

W

walk_tree() (*built-in function*), 64

Z

zipjoin() (*built-in function*), 63
zipjoin_left() (*built-in function*), 64
zipjoin_longest() (*built-in function*), 64