University of Southampton

Edward Wray ew7g23
Kavya Selvakumar ks2g23
Arezo Sanayee as17n23

# COMP2212 Programming Language Concepts (2025)

## Group 72: PCQL (Pipelined CSV Query Language) User Guide

Submission date: 15/05/2025

# Contents

# 1 Introduction

PCQL (Pipelined CSV Query Language) is a domain-specific language designed for streamlined manipulation and querying of CSV (Comma-Separated Values) files. It allows users to perform operations like filtering, selecting, grouping, merging, sorting, and modifying data in a declarative pipeline-style syntax.

This simple tool is perfect for programmers and data analysts who often work with CSV files and want to avoid the overhead of complex data processing frameworks or languages when performing simple structural operations. The pipeline approach enables users to follow the step-by-step flow of their data processing.

# 2 Design Philosophy

The language is designed to provide users with an intuitive sequential structure, which stays true to the underlying operations whilst attempting to implement a wide range of operations to suit all use cases. Here is a summary of the language's core values:

- **Simplicity:** The syntax is kept minimal and readable, suitable for users with basic programming experience.
- **Pipeline Model:** Queries are structured as a pipeline of sequential operations, where the output of each operation becomes the input to the next. This makes it easy to follow the logic and compose transformations.
- **Bare Bones:** The goal is to keep the user close to what they are doing with few abstractions. For example, there are no variable names or aliases, users specify columns with their indexes.
- **String Values:** The language only considers the values in the CSV as strings, and is mainly concerned with structuring and grouping CSVs. More complex operations like complex logic and maths is more tailored to more complex languages such as SQL or Python Pandas.

# 3 Influences

PCQL draws inspiration from the following well-established languages:

- **SQL:** Many of the operations like coalesce and merge are inspired by SQL's similar operations. Filter, select, and group-by operations are similar to SQL's approach to data manipulations.
- **Haskell:** The pipeline approach and arrow syntax is inspired by Haskell's Do-Blocks. Furthermore the declarative approach, of specifying what you want not how to do it, aligns well with Haskell's functional nature

# 4 Query Format and Syntax

A PCQL query is written in a .cql file using the following format (where [...] means optional):

```
from "input.csv" [, "input2.csv"] [raw] [to "output.csv"] do
operation1 ->
operation2 ->
...
operationN
```

**Keywords:**

- `from`: specifies the input CSV file. We can optionally specify a second file to include, however this will force a merge operation.
- `raw` (optional): indicates that the input file does not contain headers. The language by default expects the first row of the CSV to be the labels rather than the actual data. If we are using raw data, then the raw keyword adds default headers (0,1,...,n) to ensure consistent operations, which are then removed when the table is outputted.
- `to`: specifies an output CSV file. This is optional, as all queries print to standard output aswell.
- `do`: begins the list of operations.
- `->`: separates pipelined operations.

## 4.1 Example Basic Query:

```
from "students.csv" to "filteredStudents.csv" do
filter 2 == "Computer Science"
-> sort 1 asc
-> select 0,1,3
```

This query reads data from

`"students.csv"`

filters rows where column 2 is equal to "Computer Science", sorts the results by column 1 in ascending order, selects columns 0, 1, and 3, and writes the output to

`"filteredStudents.csv"`

# 5 Key Operations

## 5.1 Merge Operations

When two input files are specified, the first operation in the pipeline must be a merge specifying how to combine them into one table. It is important to note that the order in which files are specified is significant, with the first file becoming the "left" file and the second becoming the "right". All merges combine both csv files into one large csv appending the columns of the right csv to the left csv, however the type of merge decides what the rows become. There are 5 types of merge operations:

- `Inner Merge`: Only includes rows where the condition is satisfied.

```
innerMerge 0 == 2
-- Comment: "0 == 2" is the condition that column 0 is equal to column 2
```

- `Left Merge`: Includes all rows from the left file, adds the corresponding data from the right file if the condition is met for each row.

```
leftMerge 0 == 2
```

- `Right Merge`: Includes all rows from the right file, adds the corresponding data from the left file if the condition is met for each row.

```
rightMerge 0 == 2
```

- `Outer Merge`: Includes all rows from both files, fills missing data where the condition is met. Empty otherwise.

```
outerMerge 0 == 2
```

- `Cartesian Product`: Creates one large table with every possible pair that can be formed between the two tables (the same as a mathematical cartesian product). This operation does not use a condition.

```
cartesianProduct
```

## 5.2 GroupBy

GroupBy takes a column and an aggregate operation as arguments. The operation will find all rows which have a matching value in the specified column and put them in the same group. The aggregate operation will then reduce all rows into one value for each group. This then becomes the table outputted. There are three aggregate functions:

- `Count`: Counts how many rows are in the group

```
groupBy 2 count
```

- `Concat`: Concatenates all the rows into one long string

```
groupBy 3 concat
```

- `ConcatDist`: Concatenates all the distinct rows into one long string

```
groupBy 1 concatDist
```

## 5.3 Map

Map allows the user to map certain string related functions over an entire column Map takes two arguments, the target column, and the function as a string. The functions map can take are as follows:

- `upper`: Converts strings to uppercase

```
map 0 upper
```

- `lower`: Converts strings to lowercase

```
map 1 lower
```

- `trim`: Removes all leading and trailing whitespace

```
map 2 trim
```

- `length`: Outputs the length of the string

```
map 3 length
```

- `reverse`: Reverses string

```
map 0 reverse
```

- `concat "string"`: Concatenates a string to the end

```
map 1 concat "active"
```

## 5.4 Coalesce Columns

For each row, this operation examines the first `n` columns and the next `n` columns (i.e., columns `0` to `n-1` and `n` to `2n-1`). For each pair of corresponding cells, it retains the value from the first column if it is non-empty, otherwise it substitutes the value from the second column. This is especially useful after merge operations to clean up overlapping or missing data. E.g:

```
leftMerge 0 == 0 -> coalesceColumns 3
```

## 5.5 Sort

There are three main sorting operations:

- `sortLex`: Sorts the entire table in lexiographic order.

  ```
  sortLex
  ```

- `sort int asc`: Sorts the table by a specific column in ascending order.

  ```
  sort 2 asc
  ```

- `sort int desc`: Sorts the table by a specific column in descending order.

  ```
  sort 1 desc
  ```

## 5.6 Filter

Filter gets rid of any rows in the table which do not match a condition. It takes a single condition as an argument.

Conditions can be of the following:

- `==`: Tests for equality
- `!=`: Tests for inequality
- `condition && condition`: AND operation of two conditions
- `condition || condition`: OR operation of two conditions
- `!condition`: NOT operation of two conditions

When using == and !=, the left argument is always the index of the column we are comparing, whereas the right argument can either be another column to compare to (e.g: 0 == 3) or a string literal (e.g: 0 == "Doctor").

## 5.7 Select

The **select** command specifies which columns should be included in the results table. It requires **one argument**: A list of **column indices**.
**For example:**

```
select 0,1
```

This query keeps **columns 0 and 1** in the results table.
**Note:** Providing no indices will **throw an error**.

## 5.8 Drop

Similar to the select command, the **drop** command specifies which columns should **not** be included in the results table. It requires **one argument**: a list of **column indices**.
**For example:**

```
drop 1
```

This query excludes **column 1** from the results table.
**Note:** Providing no indices will **throw an error**.

## 5.9 Rename

The **Rename** command changes the header name of a specified column. It takes **two arguments**: the **column index** and the **new name**.
**For Example:**

```
rename 2 "Age"
```

This query changes the header of column 2 to "Age".
**Note:** Providing fewer than 2 arguments will **throw an error**.

## 5.10 Set

The **set** command updates a specific cell value in the table. It requires **three arguments**: **row index**, **column index**, and the **new value**.
**For example:**

```
set 2 1 "2"
```

This query will update the value at **row 2, column 1** to `"2"`. If the cell does not exist, it will be added to the table.
**Note:** Providing fewer than 3 arguments will **throw an error**.

## 5.11 AppendRow

**AppendRow** adds a new row to the end of the table. It requires a **list of string values as arguments**, which do not need to match the number of columns in the table.
**For example:**

```
appendRow "ciara" "caterpillar" "aaron"
```

This query will add a new row with those values to the end of the table

## 5.12 AddColumn

The **AddColumn** command adds a new column to the table. It takes **exactly two arguments**: the **name of the new column** and a **default value** that will be assigned to all existing rows.
**For example:**

```
addColumn "status" "active"
```

This query adds a new column named `"status"` to the table, and all existing rows will have the value `"active"` in this column.
**Note:** Providing fewer/more than **two** arguments will **throw an error**.

### 5.13 RemovePadding

RemovePadding immediately removes all trailing and leading white space from every entry in the csv file.

```
removePadding
```

# 6 Extensions

The following language extensions go beyond the core coursework specification. They enhance PCQL by adding meaningful functionality, improving both usability and flexibility.

- **Column Labels:**
  PCQL supports column headers to indicate the meaning of each column. By default, the first row in a CSV file is treated as a header row. To override this behaviour for files containing raw, unlabelled data, the `raw` keyword is used. When `raw` is specified, PCQL automatically assigns default numerical headers (0, 1, 2, ...) for internal processing. These headers are stripped from the output to preserve original structure.

- **Comments:**
  PCQL supports single-line comments to aid readability and documentation. Any line beginning with `--` is treated as a comment and ignored by the interpreter:

  ```
  -- This is a single-line comment
  ```

- **Output to Files:**
  PCQL includes explicit support for file output, enabling processed tables to be written directly to disk:

  ```
  from "input.csv" to "output.csv" do
  ...
  ```

  - The `to` keyword specifies the destination file.
  - If omitted, the output is only printed to the command line.

- **GroupBy:**
  This construct enables grouping of rows by column values to support aggregation and summary operations. Grouping functionality is described in more detail in the **Key Operations** section.

- **Fine-Grained Table Manipulation:**
  PCQL offers operations such as `set`, `appendRow`, and `addColumn` for precise and flexible modifications of table structure and content.

- **String Operations:**
  The `map` function supports built-in string transformations. Available operations include `upper`, `lower`, `reverse`, and `append "String"` among others.