qStudio/PRQL Quick Start #5 - Practical Example

Let's try a practical example of using PRQL with qStudio.

The `LymeOldToNew` table of the *Property_in_Lyme.sqlite* database has all the parcels in Lyme, showing their old and new assessments for each of the years 2022, 2023, and 2024. (In fact, there were two sets of 2024 data, one from August, and another in October that corrected many values.)

Question How can we understand what properties changed value between the August and October 2024 readings?

The following steps show the iterative process for reviewing and massaging data sets to produce the desired information. As you work through the steps, you may find oddities in the underlying data that need to be tidied up. The beauty of the PRQL lanuage (as opposed to using SQL) is that you can focus on the broad picture and let PRQL take care of the details.

Examine the `LymeOldToNew` table

Let's first review the data that we have available. (Be sure to retrieve a fresh copy of the *Property_In_Lyme.sqlite* database from the repository and add it to qStudio.)

- Click the `LymeOldToNew` table in the upper-left corner of qStudio
- Notice that it has columns for address, owner, old and new values, and a year and CollectedOn column. (There are other columns that can mostly be ignored.)
- We can use those last two columns to make comparisons between data from different years by creating a table that contains only rows where the year is 2022, or where the CollectedOn column has a certain date.

Create a table for the August 2024 rows

PRQL has the ability to create a new table from an existing one. The `let` statement does this. It assigns a name to a set of transforms that result in a table.

To start, create a new PRQL query. Always begin like this:

1. Create a new query file. Within qStudio, choose **File -> New**, save it as `AugToOct.prql` on the Desktop. (The `.prql` suffix is required so that qStudio treats the lines as a PRQL query.)
2. Add the following lines

3. Execute the query (with Cmd-E or Ctl-E)

```
let aug24 = (
   from LymeOldToNew
)
from aug24
```

At this point, the `aug24` table has all the rows of its parent table. Let's keep only the rows that were collected on August 29, 2024. To do this, add the new line below. The `@2024-08-29` is special notation for representing a date.

```
let aug24= (
    from LymeOldToNew
    filter (LO_CollectedOn == @2024-08-29)
)
from aug24
```

Now the `aug24` table contains only rows where the `LO_CollectedOn` column has that date.

We don't want most of the columns - let's select the ones we want:

```
let aug24= (
    from LymeOldToNew
    filter (LO_CollectedOn == @2024-08-29)
    select {
        PID = LO_PID,
        Location = LO_Location,
        AugOld = LO_OldValue,
        AugNew = LO_NewValue,
    }
)
from aug24
```

Now the `aug24` table contains only four columns, and only the rows from August 29, 2024. Keeping the `PID` column is especially important - we'll see why in a moment.

You'll also see that the column names in the result no longer have the "LO_" prefix because we assigned new names to them: `PID`, `Location`, `AugOld`, and `AugNew`.

Create an October table

Using a similar process, we can create a table that holds the October 2024 rows. Here's the completed example.

```
let oct24 = (
    from LymeOldToNew
    filter (LO_CollectedOn == @2024-10-31)
    select {
        LO_PID,
        OctOld = LO_OldValue,
        OctNew = LO_NewValue,
    }
)
from oct24
```

The `oct24` table has just three columns: its `PID` (we left its name as "LO_PID") as well as the old and new values. (You will probably have to comment-out (with Cmd-/ or Ctl-/) the earlier `from aug24` line.)

Use `join` to combine those tables

The `join` transform combines two tables, side-by-side, using one or more criteria to "match" the rows.

In this case, we'll use the `PID` values - they're the Parcel ID in the property tax database that uniquely identifies each parcel. Here's what we'll add to the query:

```
from aug24
join oct24 (PID == LO_PID)
```

The way to think about the above is, "Start with the `aug24` table, and `join` ("combine") it with the rows of the `oct24` table, using the values of each table's PID columns to determine which of the rows to connect."

The process the computer follows is tedious, but straightforward: Start with the first row of `aug24` - find its PID (it happens to be 1). Find a corresponding row (with a LO_PID of 1) in `oct24` and place both rows side-by-side in the result. Then examine each subsequent row of `aug24` and combine with a corresponding row of `oct24` where their PID values match. The resulting table has seven columns: the four from `aug24` and three from `oct24`

Try this now: Type the two lines above into qStudio, below the two table definitions and execute the query. (Comment out any earlier `from...` lines.)

We're now starting to get useful data. Notice the following: the values in the PID columns match; the `AugOld`

and `OctOld` values always match; and that sometimes the `AugNew` and `OctNew` values don't.

Clean up the data a little bit

As you scroll through the results, you'll notice a lot of lines that are blank, or have only `OctOld` and `OctNew` values. These result from somewhat poor data preparation. (Those lines happen to be the totals of the old and new columns that were mistakenly included in the raw data files.)

It might be possible to go back to the source data, clean it up, and update the database. But sometimes, that's not possible: the data comes from from a source that cannot be changed.

But there's another way: we can just adjust the query to leave out those lines. In this case, all the PID values are empty. Add this statement to the bottom of the query:

```
filter (aug24.PID != '')
```

This transform means, "filter (and pass through) all the PID values that are not blank ('). Notice that all those nearly-blank lines are gone.

Note: The `aug24.PID` notation is a way to specify the column named `PID` that came from the `aug24` table. (Tables of a `join` may well have identical column names - using the notation `tablename`.`columnname` helps indicate which column to use from the desired table.)

Find "new" values that *don't* match

It's hard to scan a table of numbers, comparing values in different columns. The computer can do it for us. We can make a new column using the `derive` transform that represents the *difference* between two columns.

Add this line to the end of the query:

```
derive AugToOctDiff = OctNew - AugNew
```

This creates a new column `AugToOctDiff` that contains the difference between the two columns named there.

That does half the job: the results show *lots* of rows where the difference is zero - that is, where the August and October values are the same. Let's retain only the rows where the difference is non-zero.

```
filter (AugToOctDiff != 0)
```

This result retains only the rows where the difference is non-zero. But again, it's hard to make sense of the information, since the rows seem to be in a random order. Let's sort by the difference column:

```
sort { AugToOctDiff }
```

The result now shows the property value changes, sorted by the difference between August and October.

A little more "data cleaning"

The result also has a `LO_PID` column left over from the `oct24` table. It was critical to performing the `join` operation, but is no longer needed. It's also confusing to others reading the results ("What's this `LO_PID` thing? Do I need to know about it?") Better to take it out.

There's a way to use the `select` transform to *exclude* the named columns instead of retaining them. Place a `!` in front of the tuple/list to designate the columns to exclude.

```
select !{ LO_PID }
```

This removes the `LO_PID` column, while retaining all the other columns.

Summary

This exercise shows how qStudio and PRQL allow you to investigate a data set interactively. You can view the full data set, then `filter` the rows, `select` the columns, and `join` rows together to get a result.