This article introduces the data\_algebra : a data processing tool family available in R and Python. These tools are designed to transform data either in-memory or on remote databases.

In particular we will discuss the Python implementation (also called data\_algebra) and its relation to the mature R implementations (rquery and rqdatatable).

**Introduction**

Parts of the article are in early development (and not yet ready for production use), and other parts are mature and have been used in production.

The article intent is to realize a modern data processing language based on [Codd’s relational operators](https://en.wikipedia.org/wiki/Relational_model) that is easy to maintain, has helpful tooling, and has very similar realizations (or dialects) for:

* [SQL](https://en.wikipedia.org/wiki/SQL) databases accessed from [Python](https://www.python.org/)
* Pandas Data.Frame objects in Python.
* SQL databases access from [R](https://www.r-project.org/).
* [data.table](http://r-datatable.com/) objects in R.

The idea is the notation should look idiomatic in each language. Working in Python should feel like working in Python, and working in R should feel like working in R. The data semantics, however, are designed to be close to the SQL realizations (given the close connection of SQL to the relational algebra; in particular row numbering starts at 1 and row and column order is not preserved except at row-order steps or select-columns steps respectively). The intent is: it should be very easy to use the system in either Python or R (a boon to multi-language data science projects) and it is easy to port either code or experience from one system to another (a boon for porting projects, or for data scientists working with more than one code base or computer language).

Related work includes:

* [Codd’s relational algebra](https://en.wikipedia.org/wiki/Relational_algebra)
* [SQL](https://en.wikipedia.org/wiki/SQL)
* [data.table](https://cran.r-project.org/package=data.table)
* [LINQ](https://en.wikipedia.org/wiki/Language_Integrated_Query)
* [dplyr](https://cran.r-project.org/package=dplyr)
* [dtplyr](https://cran.r-project.org/package=dtplyr)
* [table.express](https://github.com/asardaes/table.express)
* [Pandas](https://pandas.pydata.org/)
* [SQLAlchemy](https://www.sqlalchemy.org/)

The data\_algebra principles include:

* Writing data transforms as a pipeline or method-chain of many simple transform steps.
* Treating data transform pipelines or directed acyclic graphs (DAGs) as themselves being sharable data.
* Being able to use the same transform specification many places (in memory, on databases, in R, in Python).

**Example**

Let’s start with an example in Python.

For our example we will assume we have a data set of how many points different subjects score in a psychological survey. The goal is transform the data so that we see what fraction of the subjects answers are in each category (subject to an exponential transform, as often used in [logistic regression](https://en.wikipedia.org/wiki/Logistic_regression)). We then treat the per-subject renormalized data as a probabilty or diagnosis.

The exact meaning of such a scoring method are not the topic of this note. It is a notional example to show a non-trivial data transformation need. In particular: having to normalize per-subject (divide some set of scores per-subject by a per-subject total) is a classic pain point in data-processing. In classic SQL this can only be done by joining against a summary table, or in more modern SQL with a “[window function](https://en.wikipedia.org/wiki/SQL_window_function).” We want to show by working in small enough steps this can be done simply.

**Set up**

Let’s start our Python example. First we import the packages we are going to use, and set a few options.

In [1]:

import io

from pprint import pprint

import psycopg2 # <http://initd.org/psycopg/>

import pandas # <https://pandas.pydata.org>

import yaml # <https://pyyaml.org>

import db\_helpers # <https://github.com/WinVector/data_algebra/blob/master/Examples/LogisticExample/db_helpers.py>

pandas.set\_option('display.max\_columns', None)

pandas.set\_option('display.expand\_frame\_repr', False)

pandas.set\_option('max\_colwidth', -1)

Now let’s type in our example data. Notice this is an in-memory Pandas Data.Frame.

In [2]:

d\_local = pandas.DataFrame({

'subjectID':[1, 1, 2, 2],

'surveyCategory': [ "withdrawal behavior", "positive re-framing", "withdrawal behavior", "positive re-framing"],

'assessmentTotal': [5, 2, 3, 4],

'irrelevantCol1': ['irrel1']\*4,

'irrelevantCol2': ['irrel2']\*4,

})

d\_local

Out[2]:

|  | **subjectID** | **surveyCategory** | **assessmentTotal** | **irrelevantCol1** | **irrelevantCol2** |
| --- | --- | --- | --- | --- | --- |
| **0** | 1 | withdrawal behavior | 5 | irrel1 | irrel2 |
| **1** | 1 | positive re-framing | 2 | irrel1 | irrel2 |
| **2** | 2 | withdrawal behavior | 3 | irrel1 | irrel2 |
| **3** | 2 | positive re-framing | 4 | irrel1 | irrel2 |

Let’s also copy this data to a [PostgreSQL](https://www.postgresql.org/) database. Normally big data is already in the system one wants to work with, so the copying over is just to simulate the data already being there.

In [3]:

conn = psycopg2.connect(

database="johnmount",

user="johnmount",

host="localhost",

password=""

)

conn.autocommit=True

In [4]:

db\_helpers.insert\_table(conn, d\_local, 'd')

db\_helpers.read\_table(conn, 'd')

Out[4]:

|  | **subjectid** | **surveycategory** | **assessmenttotal** | **irrelevantcol1** | **irrelevantcol2** |
| --- | --- | --- | --- | --- | --- |
| **0** | 1.0 | withdrawal behavior | 5.0 | irrel1 | irrel2 |
| **1** | 1.0 | positive re-framing | 2.0 | irrel1 | irrel2 |
| **2** | 2.0 | withdrawal behavior | 3.0 | irrel1 | irrel2 |
| **3** | 2.0 | positive re-framing | 4.0 | irrel1 | irrel2 |

Normally one does not read data back from a database, but instead materializes results in the database with SQL commands such as CREATE TABLE tablename AS SELECT ....  
Also note: case in columns is a bit of nightmare. It is often best to lower-case them all.

**Back to the data\_algebra**

Now we continue our example by importing the data\_algebra components we need.

In [5]:

from data\_algebra.data\_ops import \* # <https://github.com/WinVector/data_algebra>

import data\_algebra.env

import data\_algebra.yaml

import data\_algebra.PostgreSQL

# set some things in our environment

\_, \_1, \_2, \_get = [None, None, None, lambda x: x] # don't look unbound

data\_algebra.env.push\_onto\_namespace\_stack(locals())

# ask YAML to write simpler structures

data\_algebra.yaml.fix\_ordered\_dict\_yaml\_rep()

db\_model = data\_algebra.PostgreSQL.PostgreSQLModel()

Now we use the data\_algebra to define our processing pipeline: ops. We are writing this pipeline using a [method chaining](https://en.wikipedia.org/wiki/Method_chaining) notation where we have placed Python method-dot at the end of lines using the .\ notation. This notation will look *very* much like a [pipe](https://en.wikipedia.org/wiki/Pipeline_(Unix)) to R/[magrittr](https://cran.r-project.org/package=magrittr) users.

In [6]:

scale = 0.237

ops = TableDescription('d',

['subjectID',

'surveyCategory',

'assessmentTotal',

'irrelevantCol1',

'irrelevantCol2']) .\

extend({'probability': '(assessmentTotal \* scale).exp()'}) .\

extend({'total': 'probability.sum()'},

partition\_by='subjectID') .\

extend({'probability': 'probability/total'}) .\

extend({'row\_number':'\_row\_number()'},

partition\_by=['subjectID'],

order\_by=['probability', 'surveyCategory'],

reverse=['probability']) .\

select\_rows('row\_number==1') .\

select\_columns(['subjectID', 'surveyCategory', 'probability']) .\

rename\_columns({'diagnosis': 'surveyCategory'}) .\

order\_rows(['subjectID'])

For a more pythonic way of writing the same pipeline we can show how the code would have been formatted by [black](https://github.com/psf/black).

In [7]:

py\_source = ops.to\_python(pretty=True)

print(py\_source)

TableDescription(

table\_name="d",

column\_names=[

"subjectID",

"surveyCategory",

"assessmentTotal",

"irrelevantCol1",

"irrelevantCol2",

],

).extend({"probability": "(assessmentTotal \* 0.237).exp()"}).extend(

{"total": "probability.sum()"}, partition\_by=["subjectID"]

).extend(

{"probability": "probability / total"}

).extend(

{"row\_number": "\_row\_number()"},

partition\_by=["subjectID"],

order\_by=["probability", "surveyCategory"],

reverse=["probability"],

).select\_rows(

"row\_number == 1"

).select\_columns(

["subjectID", "surveyCategory", "probability"]

).rename\_columns(

{"diagnosis": "surveyCategory"}

).order\_rows(

["subjectID"]

)

In either case, the pipeline is read as a sequence of operations (top to bottom, and left to right). What it is saying is:

* We start with a table named “d” that is known to have columns “subjectID”, “surveyCategory”, “assessmentTotal”, “irrelevantCol1”, and “irrelevantCol2”.
* We produce a new table by transforming this table through a sequence of “extend” operations which add new columns.
  + The first extend computes probability = exp(scale\*assessmentTotal), this is similar to the inverse-link step of a logistic regression. We assume when writing this pipleline we were given this math as a requirement.
  + The next few extend steps total the probabilty per-subject (this is controlled by the partition\_by argument) and then rank the normalized probabilities per-subject (grouping again specified by the partition\_by argument, and order contolled by the order\_by clause).
* We then select the per-subject top-ranked rows by the select\_rows step.
* And finally we clean up the results for presentation with the select\_columns, rename\_columns, and order\_rows steps. The names of these methods are intedned to evoke what they do.

The point is: each step is deliberately so trivial one can reason about it. However the many steps in sequence do quite a lot.

**SQL**

Once we have the ops object we can do quite a lot with it. We have already exhibited the pretty-printing of the pipeline. Next we demonstrate translating the operator pipeline into SQL.

In [8]:

sql = ops.to\_sql(db\_model, pretty=True)

print(sql)

SELECT "probability",

"subjectid",

"diagnosis"

FROM

(SELECT "probability",

"subjectid",

"surveycategory" AS "diagnosis"

FROM

(SELECT "probability",

"surveycategory",

"subjectid"

FROM

(SELECT "probability",

"surveycategory",

"subjectid"

FROM

(SELECT "probability",

"surveycategory",

"subjectid",

ROW\_NUMBER() OVER (PARTITION BY "subjectid"

ORDER BY "probability" DESC, "surveycategory") AS "row\_number"

FROM

(SELECT "surveycategory",

"subjectid",

"probability" / "total" AS "probability"

FROM

(SELECT "probability",

"surveycategory",

"subjectid",

SUM("probability") OVER (PARTITION BY "subjectid") AS "total"

FROM

(SELECT "surveycategory",

"subjectid",

EXP(("assessmenttotal" \* 0.237)) AS "probability"

FROM

(SELECT "assessmenttotal",

"surveycategory",

"subjectid"

FROM "d") "sq\_0") "sq\_1") "sq\_2") "sq\_3") "sq\_4"

WHERE "row\_number" = 1 ) "sq\_5") "sq\_6") "sq\_7"

ORDER BY "subjectid"

The SQL can be hard to read, as SQL expresses composition by inner-nesting (inside SELECT statements happen first). The operator pipeline expresses composition by sequencing or method-chaining, which can be a lot more legible. However the huge advantage of the SQL is: we can send it to the database for execution, as we do now.

Also notice the generate SQL has applied query narrowing: columns not used in the outer queries are removed from the inner queries. The “irrelevant” columns are not carried into the calculation as they would be with a SELECT \*. This early optimization comes in quite handy.

In [9]:

db\_helpers.read\_query(conn, sql)

Out[9]:

|  | **probability** | **subjectid** | **diagnosis** |
| --- | --- | --- | --- |
| **0** | 0.670622 | 1.0 | withdrawal behavior |
| **1** | 0.558974 | 2.0 | positive re-framing |

What comes back is: one row per subject, with the highest per-subject diagnosis and the estimated probabilty. Again, the math of this is outside the scope of this note (think of that as something coming from a specification)- the ability to write such a pipeline is our actual topic.

The hope is that the data\_algebra pipeline is easier to read, write, and maintain than the SQL query. If we wanted to change the calculation we would just add a stage to the data\_algebra pipeline and then regenerate the SQL query.

**Pandas**

An advantage of the pipeline is it can also be directly used on Pandas DataFrames. Let’s see how that is achieved.

In [10]:

ops.eval\_pandas({'d': d\_local})

Out[10]:

|  | **subjectID** | **diagnosis** | **probability** |
| --- | --- | --- | --- |
| **0** | 1 | withdrawal behavior | 0.670622 |
| **1** | 2 | positive re-framing | 0.558974 |

eval\_pandas takes a dictionary of Pandas DataFrames (names matching names specified in the pipeline) and returns the result of applying the pipeline to the data using Pandas commands. Currently our Pandas implementation only allows very simple window functions. This is why did’t write probability = probability/sum(probability), but instead broken the calculation into multiple steps by introducing the total column (the SQL realizaition does in fact support more complex window functions). This is a small issue with the grammar: but our feeling encourange simple steps is in fact a good thing (improves debugabbility), and in SQL the query optimizers likely optimize the different query styles into very similar realizations anyway.

**Export/Import**

Because our operator pipeline is a Python object with no references to external objects (such as the database connection), it can be saved through standard methods.”

However, data\_algebra also supports exporting a pipeline to and from simple structures that are in turn optimized for conversion to [YAML](https://yaml.org/). The simple structure format is particularly useful for writing more data\_algebra tools (such as pipeline analysis and presentation tools). And the YAML tooling makes moving a processing pipeline to another a language (such as R) quite easy.

We will demonstrate this next.

In [11]:

# convert pipeline to simple objects

objs\_R = ops.collect\_representation(dialect='R')

# print these objects

pprint(objs\_R)

[OrderedDict([('op', 'TableDescription'),

('table\_name', 'd'),

('qualifiers', {}),

('column\_names',

['subjectID',

'surveyCategory',

'assessmentTotal',

'irrelevantCol1',

'irrelevantCol2']),

('key', 'd')]),

OrderedDict([('op', 'Extend'),

('ops', {'probability': 'exp(assessmentTotal \* 0.237)'}),

('partition\_by', []),

('order\_by', []),

('reverse', [])]),

OrderedDict([('op', 'Extend'),

('ops', {'total': 'sum(probability)'}),

('partition\_by', ['subjectID']),

('order\_by', []),

('reverse', [])]),

OrderedDict([('op', 'Extend'),

('ops', {'probability': 'probability / total'}),

('partition\_by', []),

('order\_by', []),

('reverse', [])]),

OrderedDict([('op', 'Extend'),

('ops', {'row\_number': 'row\_number()'}),

('partition\_by', ['subjectID']),

('order\_by', ['probability', 'surveyCategory']),

('reverse', ['probability'])]),

OrderedDict([('op', 'SelectRows'), ('expr', 'row\_number == 1')]),

OrderedDict([('op', 'SelectColumns'),

('columns', ['subjectID', 'surveyCategory', 'probability'])]),

OrderedDict([('op', 'Rename'),

('column\_remapping', {'diagnosis': 'surveyCategory'})]),

OrderedDict([('op', 'Order'),

('order\_columns', ['subjectID']),

('reverse', []),

('limit', None)])]

In the above data structure the recursive operator steps have been linearized into a list, and simplified to just ordered dictionaries of a few defining and derived fields. In particular, the key field of the TableDescription nodes is the unique identifier for the tables, two TableDescription with the same key are referring to the same table.

We can then write this representation to YAML format.

In [12]:

# convert objects to a YAML string

dmp\_R = yaml.dump(objs\_R)

# write to file

with open("pipeline\_yaml.txt", "wt") as f:

print(dmp\_R, file=f)

**R**

This pipeline can be loaded into R and used as follows.

In [13]:

%load\_ext rpy2.ipython

In [14]:

%%R

library(yaml)

library(wrapr)

library(rquery)

library(rqdatatable)

source('R\_fns.R')

r\_yaml <- yaml.load\_file("pipeline\_yaml.txt")

r\_ops <- convert\_yaml\_to\_pipleline(r\_yaml)

cat(format(r\_ops))

table(d;

subjectID,

surveyCategory,

assessmentTotal,

irrelevantCol1,

irrelevantCol2) %.>%

extend(.,

probability := exp(assessmentTotal \* 0.237)) %.>%

extend(.,

total := sum(probability),

p= subjectID) %.>%

extend(.,

probability := probability / total) %.>%

extend(.,

row\_number := row\_number(),

p= subjectID,

o= "probability" DESC, "surveyCategory") %.>%

select\_rows(.,

row\_number == 1) %.>%

select\_columns(.,

subjectID, surveyCategory, probability) %.>%

rename(.,

c('diagnosis' = 'surveyCategory')) %.>%

orderby(., subjectID)

The above representation is nearly “R code” (it is not actually executable, unlike the Python representation, but very similar to the actual rquery steps) written using wrapr dot pipe notation. However, it can be executed in R.

In [15]:

%%R

d\_local <- build\_frame(

"subjectID", "surveyCategory" , "assessmentTotal", "irrelevantCol1", "irrelevantCol2" |

1L , "withdrawal behavior", 5 , "irrel1" , "irrel2" |

1L , "positive re-framing", 2 , "irrel1" , "irrel2" |

2L , "withdrawal behavior", 3 , "irrel1" , "irrel2" |

2L , "positive re-framing", 4 , "irrel1" , "irrel2" )

print(d\_local)

subjectID surveyCategory assessmentTotal irrelevantCol1 irrelevantCol2

1 1 withdrawal behavior 5 irrel1 irrel2

2 1 positive re-framing 2 irrel1 irrel2

3 2 withdrawal behavior 3 irrel1 irrel2

4 2 positive re-framing 4 irrel1 irrel2

We can use the R pipeline by piping data into the r\_ops object.

In [16]:

%%R

d\_local %.>%

r\_ops %.>%

print(.)

subjectID diagnosis probability

1: 1 withdrawal behavior 0.6706221

2: 2 positive re-framing 0.5589742

And the R rquery package can also perform its own SQL translation (and even execution management).

In [17]:

%%R

sql <- to\_sql(r\_ops, rquery\_default\_db\_info())

cat(sql)

SELECT \* FROM (

SELECT

"subjectID" AS "subjectID",

"surveyCategory" AS "diagnosis",

"probability" AS "probability"

FROM (

SELECT

"subjectID",

"surveyCategory",

"probability"

FROM (

SELECT \* FROM (

SELECT

"subjectID",

"surveyCategory",

"probability",

row\_number ( ) OVER ( PARTITION BY "subjectID" ORDER BY "probability" DESC, "surveyCategory" ) AS "row\_number"

FROM (

SELECT

"subjectID",

"surveyCategory",

"probability" / "total" AS "probability"

FROM (

SELECT

"subjectID",

"surveyCategory",

"probability",

sum ( "probability" ) OVER ( PARTITION BY "subjectID" ) AS "total"

FROM (

SELECT

"subjectID",

"surveyCategory",

exp ( "assessmentTotal" \* 0.237 ) AS "probability"

FROM (

SELECT

"subjectID",

"surveyCategory",

"assessmentTotal"

FROM

"d"

) tsql\_76525498125437036191\_0000000000

) tsql\_76525498125437036191\_0000000001

) tsql\_76525498125437036191\_0000000002

) tsql\_76525498125437036191\_0000000003

) tsql\_76525498125437036191\_0000000004

WHERE "row\_number" = 1

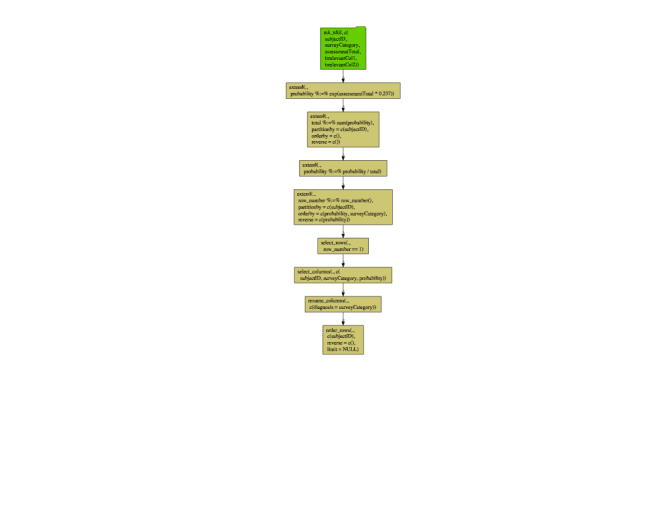
) tsql\_76525498125437036191\_0000000005

) tsql\_76525498125437036191\_0000000006

) tsql\_76525498125437036191\_0000000007 ORDER BY "subjectID"

The R implementation is mature, and appropriate to use in production. The [rquery](https://github.com/WinVector/rquery) grammar is designed to have minimal state and minimal annotations (no grouping or ordering annotations!). This makes the grammar, in my opinion, a good design choice. rquery has very good performance, often much faster than dplyr or base-R due to its query generation ideas and use of [data.table](https://cran.r-project.org/package=data.table) via [rqdatatable](https://cran.r-project.org/package=rqdatatable). rquery is a mature pure R package;.

The R implementation supports additional features such as converting a pipeline into a diagram (though that would also be easy to implement in Python on top of the collect\_representation() objects).



**Advantages of data\_algebra**

Multi-language data science is an important trend, so a cross-language query system that supports at least R and Python is going to be a useful tool or capability going forward. Obviously SQL itself is fairly cross-language, but data\_algebra adds a few features we hope are real advantages.

In addition to the features shown above, a data\_algebra operator pipeline carries around usable knowledge of the data transform. For example:

In [18]:

# report all tables used by the query, by name

ops.get\_tables()

Out[18]:

{'d': TableDescription(table\_name='d', column\_names=['subjectID', 'surveyCategory', 'assessmentTotal', 'irrelevantCol1', 'irrelevantCol2'])}

In [19]:

# report all source table columns used by the query

ops.columns\_used()

Out[19]:

{'d': {'assessmentTotal', 'subjectID', 'surveyCategory'}}

In [20]:

# what columns does this operation produce?

ops.column\_names

Out[20]:

['subjectID', 'diagnosis', 'probability']

**Conclusion**

The data\_algebra is part of a powerful cross-language and mutli-implementaiton family data manipulation tools. These tools can greatly reduce the development and maintenance cost of data science projects, while improving the documentation of project intent.

**Appendix:**

Demonstrate we can round-trip a data\_algebra through YAML and recover the code.

In [21]:

# land the pipeline as a file

objs\_Python = ops.collect\_representation()

dmp\_Python = yaml.dump(objs\_Python)

with open("pipeline\_Python.txt", "wt") as f:

print(dmp\_Python, file=f)

In [22]:

# read back

with open("pipeline\_Python.txt", "rt") as f:

ops\_text = f.read()

ops\_back = data\_algebra.yaml.to\_pipeline(yaml.safe\_load(ops\_text))

print(ops\_back.to\_python(pretty=True))

TableDescription(

table\_name="d",

column\_names=[

"subjectID",

"surveyCategory",

"assessmentTotal",

"irrelevantCol1",

"irrelevantCol2",

],

).extend({"probability": "(assessmentTotal \* 0.237).exp()"}).extend(

{"total": "probability.sum()"}, partition\_by=["subjectID"]

).extend(

{"probability": "probability / total"}

).extend(

{"row\_number": "\_row\_number()"},

partition\_by=["subjectID"],

order\_by=["probability", "surveyCategory"],

reverse=["probability"],

).select\_rows(

"row\_number == 1"

).select\_columns(

["subjectID", "surveyCategory", "probability"]

).rename\_columns(

{"diagnosis": "surveyCategory"}

).order\_rows(

["subjectID"]

)

In [23]:

# confirm we have a data\_algebra.data\_ops.ViewRepresentation

# which is the class the data\_algebra pipelines are derived from

isinstance(ops\_back, data\_algebra.data\_ops.ViewRepresentation)

Out[23]:

True

In [24]:

# be neat

conn.close()