Relation: The Missing Container

Scott James‡*, James Larkin‡

Abstract

The humble mathematical *relation*[1], a fundamental (if implicit) component in computational algorithms, is conspicuously absent in most standard container collections, including Python's. In this paper, we present the basics of a relation container, and why you might use it instead of other methods. The concept is simple to implement and easy to use. We will walk through with code examples using our implementation of a relation (https://pypi.python.org/pypi/relate)

Background: It's the Little Things

In our work in surface and aviation traffic simulation we deal with many moving pieces, terabytes of streaming information. Managing this much information pieces requires, unsurprisingly, some significant computational machinery: clusters of multiprocessors; different interworking database topologies: HDF5, NoSQL and SQL; compiled code, scripted code; COTS tools, commercial and open source code libraries. For the Python components of our work, we are fortunate to have data crunching libraries: numpy, pandas etc... However, we kept finding that, despite this wealth of machinery, we would get caught up on the little things.

There may be thousands of flights in the air at any one time, but there are far fewer *types* of aircraft. There may be millions of vehicles on the road, but only a handful of vehicle categories. Whereas we could place these mini-databases into our data crunching tools as auxiliary tables, we didn't. It didn't make sense to perform a table merge with streaming data when we could do a quick lookup, on-the-fly, when we needed to. We didn't want to create a table with ten rows and two columns when we could easily put that information into a dictionary, or a list. We didn't want to implement our transient, sparse table with a graph database or create tables with an 'other' column which we would then have to parse anyhow. And besides the traffic specific information, there were all those other pesky details: file tags, user aliases, color maps.

Instead we cobbled together our mini-databases with what we had within easy mental reach: lists, sets and dictionaries. And when we needed to do a search, or invert keys/values, or assure uniqueness of mappings, we would create a loop, a list comprehension or a helper class.

* *Corresponding author: scott.james@noblis.org*
‡ *Noblis*

After some time it occurred to us that what we were really doing with our less-than-big data was reinventing a mathematical relation, ... over and over again. Once we realized that, we replaced the bookkeeping code managing our mini-databases with relation instances. This resulted in a variety of good things: reduced coding overhead, increased clarity of purpose and, oddly, improved computational efficiency.

What is a relation and what is it good for?

A relation a simply a pairing of elements of one set, the *domain*, with another, the *range*. Rephrasing more formally, a relation is a collection of tuples (x,y) where x is in the domain and y is in the range. A relation, implemented as code, can perform a variety of common tasks:

- **Inversion**: quickly find the values(range) associated with a key(domain)
- **Partitioning**: group values into unique buckets
- **Aliasing**: maintain a unique pairing between keys and values
- **Tagging**: associate two sets in an arbitrary manner

These roughly correspond to the four *cardinalities* of a relation:

- **Many-to-one (M:1)**: a function, each range value having possibly multiple values in the domain
- **One-to-many (1:M)**: a categorization, where each element in the domain is associated with a unique group of values in the range
- **One-to-one (1:1)**: an isomorphism, where each element in the domain is uniquely identified with a single range value
- **Many-to-many (M:N)**: an unrestricted pairing of domain and range

What is it not good for?

The relation, at least as we have implemented it, is a chisel, not a jack-hammer. It is meant for the less-than-big data not the actually-big data. When computational data is well-structured, vectorized or large enough to be concerned about storage, we use existing computational and relational libraries. A relation, by contrast, is useful when the data is loosely structured, transient, and in no real danger of overloading memory.

The API

Using a relation should be easy, as easy as using any fundamental container. It should involve as little programming friction as possible. It should feel natural and familiar. To accomplish these

Method	Comment
`__init__`	establish the cardinality and ordering of a Relation
`__setitem`	assign a range element to a domain element
`__getitem`	retrieve range element(s) for a domain element
`__delitem`	remove a domain element and all associated range pairings. If the range element has no remaining pairings, delete it.
`extend`	combine two Relation objects
`values`	return the domain
`keys`	returns list of domains
`__invert__`	swap domain and range

TABLE 1

goals, we created our Relation class by inheriting and extending *MutableMapping*:

In essence, it will look and feel like a dictionary, but with some twists.

Example 1 (Many-to-many)

For example, suppose we need to map qualitative weather conditions to dates:

```
weather = Relation()
weather['2011-7-23']='high-wind'
weather['2011-7-24']='low-rain'
weather['2011-7-25']='low-rain'
weather['2011-7-25']='low-wind'
```

Note that in the last statement the assignment operator performs an append not an overwrite. So:

```
weather['2014-7-25']
```

Produces a *set* of values:

```
{'low-rain','low-wind'}
```

Relation also provides an inverse:

```
(~weather)['low-rain']
```

Also producing a set of values:

```
{'2014-7-25','2014-7-24'}
```

For our work, other many-to-many relations include:

- Flight numbers and airports
- Auto makers and vehicle classes
- Neighboring planes (or autos) at an instant of time

Cardinality

Relations look like a dictionary but also provide the ability to

1) Assign many-to-many values
2) Invert the mapping directly

Relations become even more valuable when we have the ability to enforce the degree of relationship, i.e. cardinality. As mentioned, there are four cardinalities used in the relation object class:

Many-to-one assignment is already supported by Python's built-in dictionary (minus the inversion); however, the remainder of the cardinalities are not[2].

Relationship	Shortcut	Pseudonyms
many-to-one	M:1	function, mapping, assignment
one-to-many	1:M	partition, category
one-to-one	1:1	aliasing, isomorphism
many-to-many	M:N	general

TABLE 2

Example 2 (One-to-One)

```
airport = Relation(cardinality='1:1')
airport['ATL'] = 'Hartsfield-Jackson Atlanta International'
airport['KORD'] = 'Chicago O'Hare International'
```

When the relation is forced to be 1:1, the results are no longer sets:

```
airport['ATL']
> 'Hartsfield-Jackson Atlanta International'
```

And assignments overwrite *both* the domain and the range:

```
# use the full four-letter international code ...
# not the US 3-letter code
airport['KATL'] = 'Hartsfield-Jackson Atlanta International'
airport['ATL']
> KeyError: 'ATL'
```

Note that, similar to a dictionary silently overwriting a key-value pair, a 1:1 relation silently overwrites a value-key pair, and in this case, removes the stranded key. Also worth noting, for cardinalities M:1 and 1:1, a dictionary literal can also serve as syntactic sugar for an initializer:

```
airport = Isomorphism(
    {'KATL':'Hartsfield-Jackson Atlanta International',
    'KORD':'Chicago O'Hare International'})
airport['KATL']
> 'Hartsfield-Jackson Atlanta International'
```

For our work, other 1:1 mappings include:

- User names and company id
- Automobile manufacturers and their abbreviations
- Color codes and their representations in various simulation tools (using a chain of 1:1 containers)

Comparing Relation Implementations

The relation container is fast, as fast as a dictionary. It should be; it is implemented by two dictionaries: one for each mapping direction. However, there are other ways to implement many-to-many relations. In this section we compare the relation against two other implementations: a Pandas data frame and a NetworkX graph.

Our test data will consist of 200 two-digit alphanumeric values (domain) and 1 million numeric values (range) for a total of approximately 10 million unique entries. We describe the implementations of the lookups and then compare speeds.

Data Frame

To implement a M:N relationship using a data frame, we create a two-column table:

A forward search can performed as follows:

```
df['range']==887267]['domain']
```

And a reverse search as:

```
df[df['domain']=='YR']['range']
```

Each of these searches can be accelerated by indexing.

Domain	Range
UF	423423
OP	3242
FD	887267
YR	343
...	...

TABLE 3

Method	Forward (ms)	Reverse (ms)
Pandas	7.34e2	7.94e1
Pandas (indexed)	1.97e2	7.81e-2
Graph	9.47e0	6.84e-4
Relate	3.76e-4	4.58e-4

TABLE 4

NetworkX

To implement an M:N relation using a NetworkX Graph we use a bipartite graph, that is, a graph connecting two disjoint sets, creating the relations by linking the nodes from one set (domain) to another set (range)

Both forward and reverse searches are performed in the same manner:

```
# forward, using domain nodes
G.neighbors('YR')
# reverse, using range nodes
G.neighbors(887267)
```

Timings

We collect timings using the Python's `timeit` function:

In all cases, Relate is faster, most significantly when searching on strings as opposed to numeric values. Of course, data frames and graphs have many more features than a relation. Also, the two-dictionary Relation implementation is cheating: it precomputed the only two searches it was built to handle; moreover, it did so at a cost of doubling the memory footprint. But this is precisely the use-case for which the relation was created: space at non-critical levels but economy of code and code performance crucial.

Sparse Matrix

One other implementation worth mentioning is a sparse matrix. Viewing the nonzero elements of a sparse matrix as a connection between the row (domain) and column (range) indices also produces an M:N relationship. The power of the sparse matrix is in its suitability to large-scale numerical computations. The relation container proposed, however, is designed to match general datatypes, including non-numerical. Providing a direct comparison between the two is thus somewhat difficult as the two are used for different purposes.

More Examples

The relation object is a basic concept, and as such useful in limitless contexts. A few more examples are worth mentioning.

Tags (Many-to-Many)

Over the last decade, we've seen *tags* invade our previously hierarchical organized data. Tags are now ubiquitous, attached to our: photos, files, URL bookmarks, to-do items etc ...

Tags are also exactly a many-to-many relationship:

```
files = Relation()

files['radar-2011-7-23.png'] = 'image'
files['radar-2011-7-23.png'] = 'KATL'
files['departure-procedures.doc'] = 'KATL'
files['departure-procedures.doc'] = 2015

#find the files associated with Atlanta
(~files)['KATL']
> {'radar-2011-7-23.png','departure-procedures.doc'}

# find the attributes for particular file
files['departure-procedures.doc']
> {2015,'KATL'}
```

We tag our simulation products to allow flexible retrieval and searching. With an in-code tagging scheme we can automatically attach tags at the file system level and then query these tags with both in-code and operating system level tools.

Taxonomies (One-to-Many)

We mentioned earlier that the 1:M relation is a partition, a way to categorize objects into groups. Nesting 1:M relations creates a backward-searchable taxonomy. An example in our work are en-route air traffic sectors, the nested polyhedrons through which aircraft fly:

```
sectors=Relation(cardinality='1:M')
sectors['ZNY'] = 'ZNY010'
sectors['ZNY'] = 'ZNY034'
sectors['ZNY010'] = 'ZNY010-B'
sectors['ZNY010'] = 'ZNY010-2'
sectors['ZNY034'] = 'ZNY034-B'
sectors['ZNY034'] = 'ZNY034-11'

(~sectors)['ZNY034-B']
> 'ZNY034'

(~sectors)[(~sectors)['ZNY034-B']]
> 'ZNY'
```

Using a taxonomy of sectors as above allows us to quickly access aggregate information at different granularities as the flight progresses.

When to Use What for What

Modern high-level computing languages provide us with a robust set of containers. We feel, of course, that a relation container is a valuable addition but, we also feel one should use the most economical container for the task. Asking questions about the type of data being stored and the relationship between an element and its attributes is crucial, even for the less-than-big data:

Choosing the best matching structure for your data set doesn't just help with the code, it helps with the intent, providing the next programmer touching the code with your vision of the structure, and also some safety belts in case they didn't see it the first time.

Conclusion

The relation object provides an easy-to-use invertible mapping structure supporting all four relationship cardinalities: 1:1, 1:M, M:1 and M:N. Using the relation library can simplify your

Content	Structure
unordered set of unique objects	set
ordered set of non-unique objects	list
ordered set of unique objects	OrderedDict
unidirectional mapping	dictionary
bidirectional mapping	relation
mapping with restricted cardinalities	relation
multiple, fixed attributes per element	data frame/table
variant attributes per element	relation

TABLE 5

code and eliminate the need for repeated, ad hoc patterns when managing your less-than-big working data structures.

One of the best things about the relation data container is its ease of implementation within Python. For a simple, yet complete example, see our implementation at https://pypi.python.org/pypi/relate.

1. http://www.purplemath.com/modules/fcns.htm

2. For 1:1 mapping, however we also recommend the excellent bidict package https://bidict.readthedocs.org/en/master/intro.html#intro