Dee - D in Python

Greg Gaughan

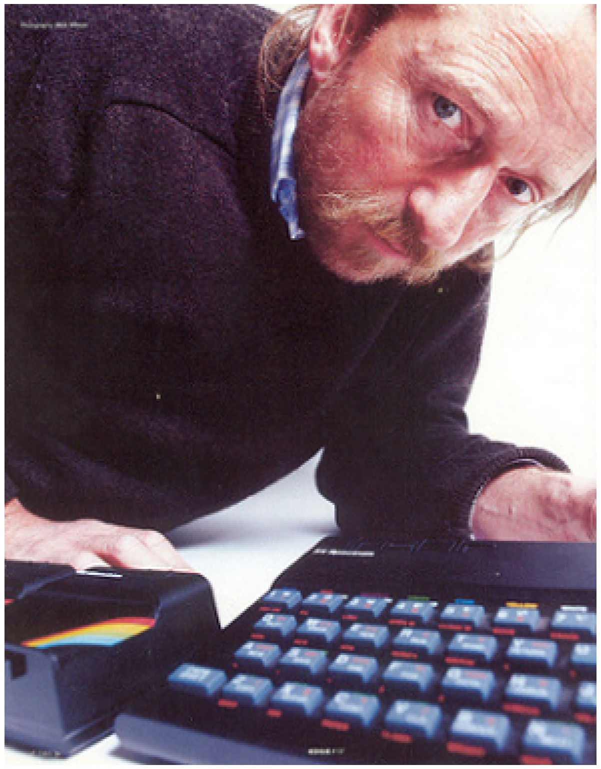
The Third Manifesto Implementers' Workshop

Northumbria University, June 2011

My Database History

1983 - Sinclair ZX Spectrum

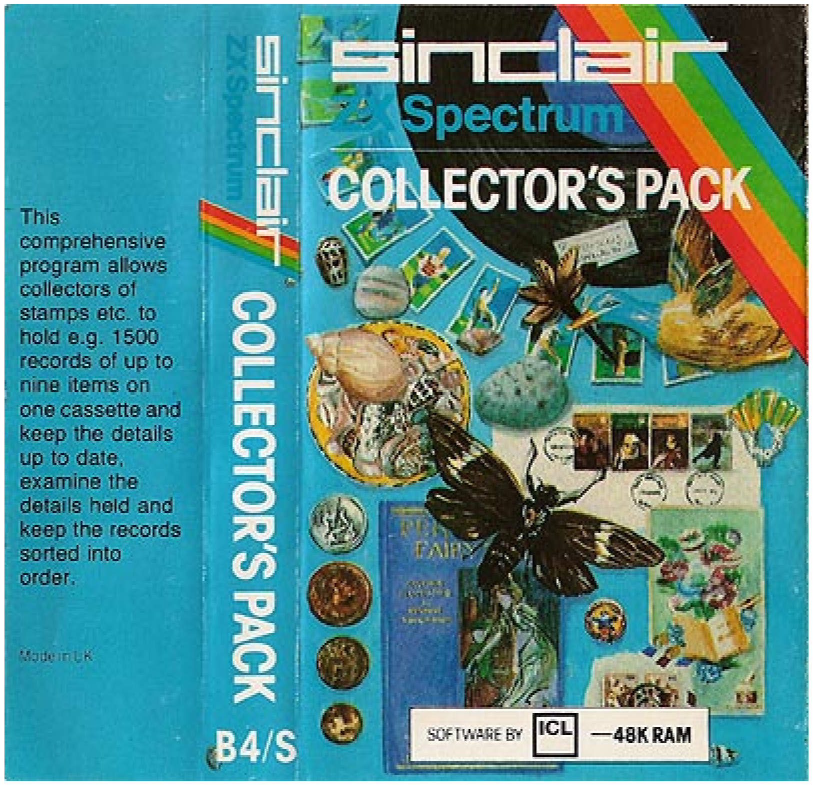
Sinclair machines were designed by Rick Dickinson, a graduate of Northumbria University



(photo from http://www.flickr.com/photos/9574086@N02/697755822/in/photostream)

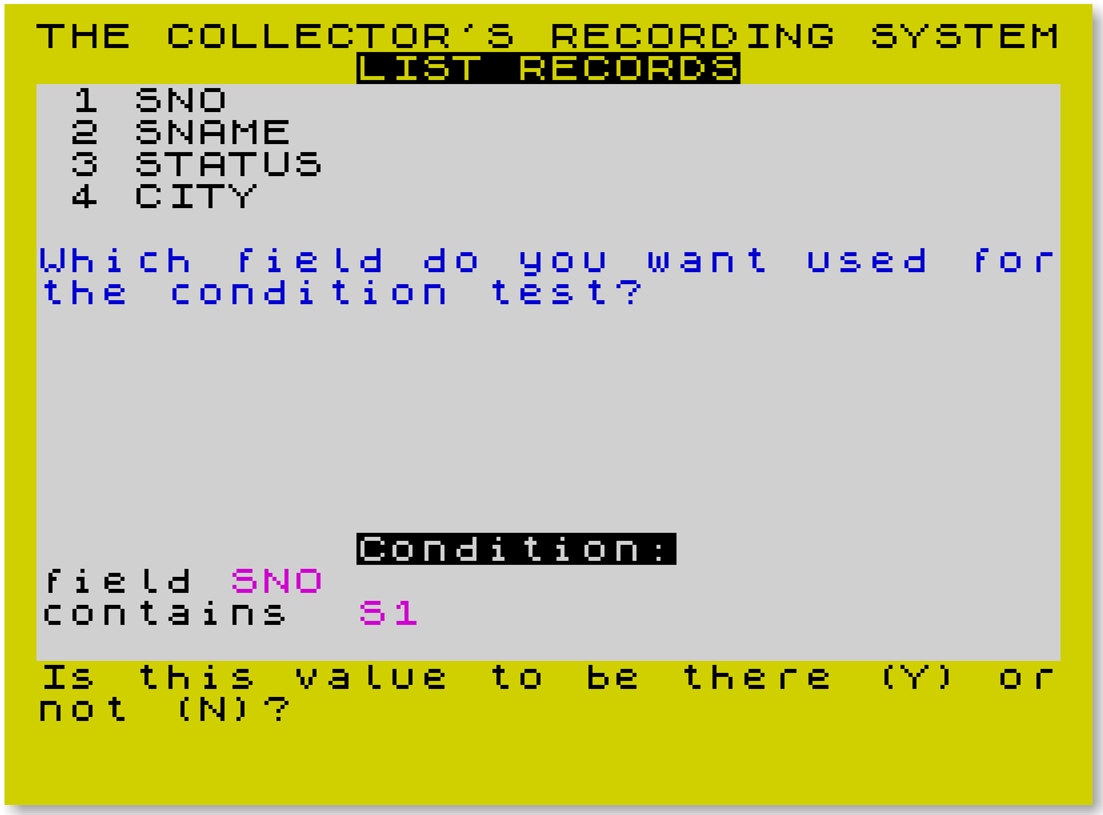
1984 - ICL's Collector's Pack

Could store "1500 records of up to 9 items"


This comprehensive program allows collectors of stamps etc. to hold e.g. 1500 records of up to nine items on one cassette and keep the details up to date, examine the details held and keep the records sorted into order.
Made in UK
sinclair ZX Spectrum
COLLECTOR'S PACK
B4/S
sinclair
ZX Spectrum
COLLECTOR'S PACK
SOFTWARE BY ICL
—48K RAM

1984 - ICL's Collector's Pack

On cassette tape (key/value pairs)



```
THE COLLECTOR'S RECORDING SYSTEM
          LIST RECORDS
  1  SNO
  2  SNAME
  3  STATUS
  4  CITY

Which field do you want used for
the condition test?





          Condition:
field SNO
contains    S1
Is this value to be there (Y) or
not (N)?
```

The Bad, the Good and the Ugly

- 1987 - ISAM and partitioned files with hash lookups (key/value pairs)
- 1991 - Oracle 7 - correlated sub-selects - Wow!
- 1994 - dBASE - procedural with local files
- 1997 - Sybase SQL Anywhere
- 1998 - Read **"A Guide to the SQL Standard"** (Date/Darwen)
- 1999 - Read **"Database Management Systems"** (Ramakrishnan)
- MySQL becoming very popular...

MySQL

From the **MySQL Reference Manual** (circa 2001 **-** my highlighting):

> 5.4.5.1 Reasons NOT to Use Foreign Keys constraints
>
> There are so many problems with foreign key constraints that we don't know where to start:
>
> - Foreign key constraints make life very complicated, because the foreign key definitions must be stored in a database and implementing them would destroy the whole ''nice approach'' of using files that can be moved, copied, and removed.
> - The speed impact is terrible for INSERT and UPDATE statements, and in this case almost all FOREIGN KEY constraint checks are useless because you usually insert records in the right tables in the right order, anyway.
> - There is also a need to hold locks on many more tables when updating one table, because the side effects can cascade through the entire database. It's MUCH faster to delete records from one table first and subsequently delete them from the other tables.
> - You can no longer restore a table by doing a full delete from the table and then restoring all records (from a new source or from a backup).

MySQL

- If you use foreign key constraints you can't dump and restore tables unless you do so in a very specific order.
- It's very easy to do ''allowed'' circular definitions that make the tables impossible to re-create each table with a single create statement, even if the definition works and is usable.
- It's very easy to overlook FOREIGN KEY ... ON DELETE rules when one codes an application. It's not unusual that one loses a lot of important information just because a wrong or misused ON DELETE rule.

The only nice aspect of FOREIGN KEY is that it gives ODBC and some other client programs the ability to see how a table is connected and to use this to show connection diagrams and to help in building applicatons *[sic]*.

Towards ThinkSQL

- 2000 - Sybase ASE
  - `sp_primarykey`, `sp_foreignkey`
  - `*=` outer join

"If you submit a query with an outer join and a qualification on a column from the inner table of the outer join, the results may not be what you expect. The qualification in the query does not restrict the number of rows returned, but rather affects which rows contain the null value. For rows that do not meet the qualification, a null value appears in the inner table's columns of those rows."

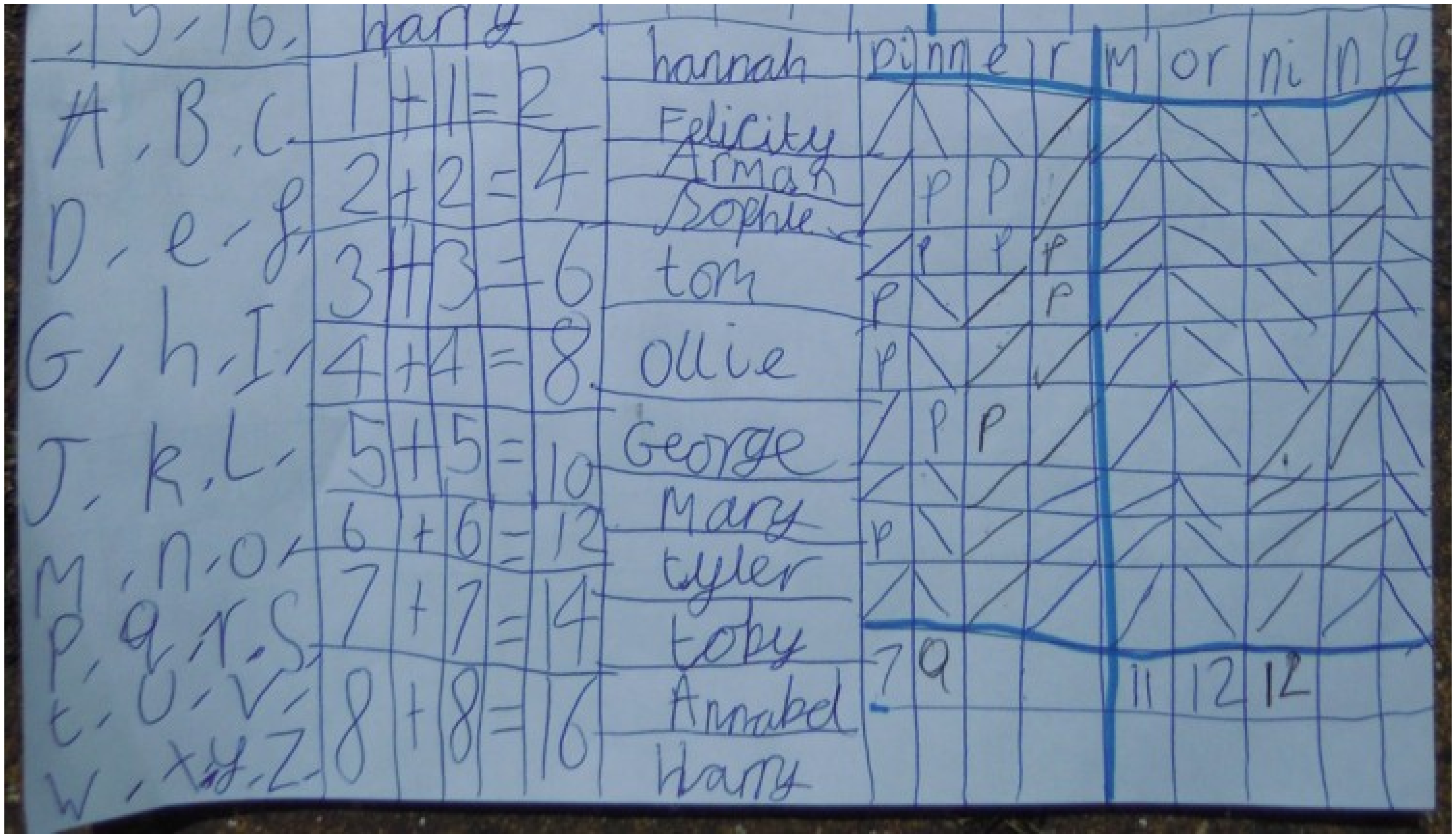
Adaptive Server Enterprise 12.5 User Guide

Towards ThinkSQL

- Led to ThinkSQL (http://www.thinksql.co.uk)
    - SQL/92 compliant (including `WHERE (a, b) = (1, 2)` and standalone `VALUES`)
    - Multi-version concurrency control (proposed by David Reed in 1983)
    - ODBC, and native JDBC and Python drivers
    - Some bootstrapping

Child Tables

One of Mary's tables:



/ 5 / 16. /	Mary		di	nne	e	r	M	or	ni	n	g
		hannah									
A, B, C.	1 + 1 = 2	Felicity	✓	✓	✓	✓	✓	✓	✓	✓	✓
D, e, f,	2 + 2 = 4	Arman	✓	p	p	✓	✓	✓	✓	✓	✓
G, h, I,	3 + 3 = 6	Sophie	✓	p	p	p	✓	✓	✓	✓	✓
J, k, L.	4 + 4 = 8.	tom	p	✓	✓	p	✓	✓	✓	✓	✓
M, n, o,	5 + 5 = 10	ollie	p	✓	✓	✓	✓	✓	✓	✓	✓
P, q, r, S	6 + 6 = 12	George	✓	p	p	✓	✓	✓	✓	✓	✓
t, U, V,	7 + 7 = 14	Mary	✓	✓	✓	✓	✓	✓	✓	✓	✓
W, x, y, Z	8 + 8 = 16	tyler	p	✓	✓	✓	✓	✓	✓	✓	✓
		toby	✓	✓	✓	✓	✓	✓	✓	✓	✓
		Annabel	7	9			11	12	12		
		Mary									

Towards Dee

- 2005 - Object-relational-mappers (ORMs) and XML becoming very popular
- 2006 - Read **"Foundation for Future Database Systems (The Third Manifesto)"** (Date/Darwen)
  - Eventually got the big abstraction: relation variables referring to sets of propositions plus algebraic operations, instead of tables as row containers to be modified and iterated over
- Led to Dee (http://www.quicksort.co.uk)
- 2011 - NOSQL becoming popular... (key/value pairs)

Why Python?

Python is interpreted, expressive, readable and powerful.

Python is strongly and dynamically typed, so variables are not pre-declared to be specific types, so type errors in Python are runtime errors (like database constraint violations). But I find the latent typing liberating when developing software and that it leads to:

- less syntactic noise, so:
    - faster development
    - more readable code
    - more debuggable code
    - improved code quality
- more flexible routines and libraries, without resorting to 'patterns' or bolt-on generics or templates

Why Python?

Python has `True`, `False` and sets built-in, e.g.

```
>>> 'b' in {'a', 'b', 'c'}
True
>>> 'd' not in {'a', 'b', 'c'}
True
>>> 'd' in {'a', 'b', 'c'}
False
>>> {'a', 'b', 'c'} | {'b', 'd'}
{'a', 'c', 'b', 'd'}
>>> {'a', 'b', 'c'} & {'b', 'd'}
{'b'}
>>> {'a', 'b', 'c'} - {'b', 'd'}
{'a', 'c'}
```

Dee

- Version 0.12 is available now
- Version 0.2 is being developed

Tuples and Relations

Python's built-in dictionaries can represent tuples, e.g.:

```
{'SNO':'S1', 'SNAME':'Smith', 'STATUS':20, 'CITY':'London'}
```

Then to represent this relation:

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

use the Dee `Relation` class:

```
Relation(['SNO', 'SNAME', 'STATUS', 'CITY'],
         [{'SNO':'S1', 'SNAME':'Smith', 'STATUS':20, 'CITY':'London'},
          {'SNO':'S2', 'SNAME':'Jones', 'STATUS':10, 'CITY':'Paris'},
          {'SNO':'S3', 'SNAME':'Blake', 'STATUS':30, 'CITY':'Paris'},
          {'SNO':'S4', 'SNAME':'Clark', 'STATUS':20, 'CITY':'London'},
          {'SNO':'S5', 'SNAME':'Adams', 'STATUS':30, 'CITY':'Athens'},
         ])
```

Tuples and Relations

or using the positional shorthand:

```
Relation(['SNO', 'SNAME', 'STATUS', 'CITY'],
         [('S1', 'Smith', 20,       'London'),
          ('S2', 'Jones', 10,       'Paris'),
          ('S3', 'Blake', 30,       'Paris'),
          ('S4', 'Clark', 20,       'London'),
          ('S5', 'Adams', 30,       'Athens'),
         ]
        )
```

Relations can be assigned to relvars, e.g.

```
S = Relation(['SNO', 'SNAME', 'STATUS', 'CITY'],
             [('S1', 'Smith', 20,       'London'),
              ('S2', 'Jones', 10,       'Paris'),
              ('S3', 'Blake', 30,       'Paris'),
              ('S4', 'Clark', 20,       'London'),
              ('S5', 'Adams', 30,       'Athens'),
             ],
             {'PK':(Key, ['SNO'])}
            )
```

Tuples and Relations

Relation values can be displayed as tables, e.g.

```
>>> print S
+-----+-------+--------+--------+
| SNO | SNAME | STATUS | CITY   |
+=====+-------+--------+--------+
| S1  | Smith | 20     | London |
| S2  | Jones | 10     | Paris  |
| S3  | Blake | 30     | Paris  |
| S4  | Clark | 20     | London |
| S5  | Adams | 30     | Athens |
+-----+-------+--------+--------+
```

And a Python representation of them can be retrieved, e.g.

```
>>> print `S`
Relation(['SNO','SNAME','STATUS','CITY'],
[{'STATUS': 20, 'SNO': 'S1', 'SNAME': 'Smith', 'CITY': 'London'},
{'STATUS': 10, 'SNO': 'S2', 'SNAME': 'Jones', 'CITY': 'Paris'},
{'STATUS': 30, 'SNO': 'S3', 'SNAME': 'Blake', 'CITY': 'Paris'},
{'STATUS': 20, 'SNO': 'S4', 'SNAME': 'Clark', 'CITY': 'London'},
{'STATUS': 30, 'SNO': 'S5', 'SNAME': 'Adams', 'CITY': 'Athens'}],
{'PK':(Key, ['SNO'])})
```

Tuples and Relations

Python's `eval` can evaluate a string as code at runtime, e.g.

```
>>> S == eval(`S`)
True
```

And there are a number of methods to load relvars from, and unload to, Python lists, e.g.

```
>>> sorted([(t.CITY, t.SNO) for t in S.to_tuple_list()])
[('Athens', 'S5'), ('London', 'S1'), ('London', 'S4'),
 ('Paris', 'S2'), ('Paris', 'S3')]
```

Algebra

Four fundamental operators from the A algebra are defined as Python functions:

- ◀AND▶
- ◀OR▶ (limited to union for practical reasons)
- ◀MINUS▶ (i.e. practical ◀NOT▶)
- ◀REMOVE▶

All the other operators were built on the fundamental ones, e.g. COMPOSE was built from ◀REMOVE▶ and ◀AND▶:

```python
def COMPOSE(r1, r2):
    """AND and then REMOVE common attributes (macro)"""
    a = r1.heading & r2.heading
    return REMOVE(AND(r1, r2), a)
```

Algebra

RESTRICT and EXTEND are implemented in terms of AND and use anonymous functions to define the expressions, e.g.:

```
RESTRICT(S, lambda t: t.STATUS == 20)
```

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

Any Python expression can be passed this way. So here, complex boolean expressions including boolean operators and function calls can be built, e.g.

```
RESTRICT(S, lambda t: 10 < t.STATUS < 30 and t.SNAME.startswith('C'))
```

SNO	SNAME	STATUS	CITY
S4	Clark	20	London

Algebra

The lambda function passed to the EXTEND function should return a tuple, so for example:

```
EXTEND(S, lambda t: {'INITIAL':t.SNAME[:1]})
```

SNO	SNAME	STATUS	CITY	INITIAL
S1	Smith	20	London	S
S2	Jones	10	Paris	J
S3	Blake	30	Paris	B
S4	Clark	20	London	C
S5	Adams	30	Athens	A

Built-in Relational Functions

- AND (`&`)
- COMPOSE
- EXTEND
- GENERATE
- GROUP / UNGROUP
- MATCHING
- MINUS (`-`)
- OR (`|`)
- QUOTA
- REMOVE
- RESTRICT
- TCLOSE
- WRAP / UNWRAP

Changes to Functions in Version 0.2

Added

- IMAGE

Removed

- SEMIJOIN (use MATCHING instead)
- SEMIMINUS (use not MATCHING instead)
- SUMMARIZE (use IMAGE instead)
- DIVIDE and DIVIDE_SIMPLE (use IMAGE instead)

Aggregate Operators

- ALL
- ANY
- AVG
- COUNT
- IS_EMPTY
- MAX
- MIN
- SUM

Relation Constants

- DEE (TRUE)
- DUM (FALSE)

Operators

The operators associated with the `Relation` class are overridden to give a natural Python-like (Pythonic) syntax, e.g.

```
r1 == r2           #equality
r1 != r2           #inequality
r1 <= r2           #subset
r1 < r2            #proper subset
r1 >= r2           #superset
r1 > r2            #proper superset
t1 in r1           #relation (and tuple) membership
t1 not in r1       #relation (and tuple) exclusion
r1(['a1', 'a2'])   #projection
r1 & r2            #AND, i.e. intersection or natural join or Cartesian join
r1 | r2            #OR, i.e. union
r1 - r2            #MINUS
```

Functions and Methods

Many functions (upper-case names) are also available as methods (lower-case names) on the `Relation` class and these can be conveniently chained together to offer an alternative syntax. For example:

```
(r1 & r2).where(lambda t: t.a1 == 5 and t.a2 < 100).remove(
          ['a3', 'a4']).group(['a6'], 'grouped')
```

would join `r1` and `r2`, then filter the result, then remove attributes `a3` and `a4`, and then add a grouping. The chaining provides the opportunity for more efficient execution using pipelining rather than fully evaluating and storing each step.

To use the functions directly, the following syntax would give the same result:

```
tr0 = AND(r1, r2)
tr1 = RESTRICT(tr0, lambda t: t.a1 == 5 and t.a2 < 100)
tr2 = REMOVE(tr1, ['a3', 'a4'])
GROUP(tr2, ['a6'], 'grouped')
```

Functions and Methods

Or the more LISPy version:

```
GROUP(
    REMOVE(
        RESTRICT(
            AND(r1, r2),
            lambda t: t.a1 == 5 and t.a2 < 100
        ),
        ['a3', 'a4']
    ),
    ['a6'],
    'grouped'
)
```

Functions and Methods

Building on the `Relation`'s extend, project and remove methods, we have:

```
def WRAP(r, Hr, wrapname):
    """Wrapping (macro)"""
    return r.extend(lambda t: {wrapname:t.project(Hr)}).remove(Hr)
```

And here's a definition of a recursive relational operator:

```
def TCLOSE(r):
    """Transitive closure (an example of a recursive relational operator)
       (macro) (not optimised for speed)
    """
    if len(r.heading) != 2:
        raise InvalidOperation("TCLOSE expects a binary relation, "
                               "e.g. with a heading ['X', 'Y']")

    _X, _Y = r.heading
    TTT = r | (COMPOSE(r, r.rename({_Y:'_Z', _X:_Y})).rename({'_Z':_Y}))
    if TTT == r:
        return TTT
    else:
        return TCLOSE(TTT)
```

Relation Valued Attributes

The GROUP function provides a shorthand for nesting relations. e.g.

```
GROUP(SP, ['PNO', 'QTY'], 'PQ')
```

SNO	PQ	
S1	**PNO**	**QTY**
	P1	300
	P2	200
	P3	400
	P4	200
	P5	100
	P6	100
S2	**PNO**	**QTY**
	P1	300
	P2	400
S3	**PNO**	**QTY**
	P2	200

SNO	PQ	
	PNO	**QTY**
S4	P2	200
	P4	300
	P5	400

Image Relations

```
S.where(lambda t: IMAGE(SP)(['PNO']) == P(['PNO']))
```

SNO	SNAME	STATUS	CITY
S1	Smith	20	London

```
S.extend(lambda t: {'TOTQD':SUM(IMAGE(SP)(['QTY']))})
```

SNO	SNAME	STATUS	CITY	TOTQD
S1	Smith	20	London	1000
S2	Jones	10	Paris	700
S3	Blake	30	Paris	200
S4	Clark	20	London	900
S5	Adams	30	Athens	0

IMAGE replaces DIVIDE and SUMMARIZE in the next version.

Virtual Relvars

Virtual relvars (views) can be created using `lambda` to defer the evaluation of relational expressions. For example:

```
r3 = r1 & r2
```

immediately evaluates the expression on the right hand side and assigns the result to r3. Whereas:

```
r3 = View(lambda: r1 & r2)
```

assigns the expression `r1 & r2` to the View `r3` which will then be re-evaluated each time the value of `r3` is needed.

The `View` class inherits from `Relation`.

Updates

Insert and delete methods are available for the `Relation` class and have been set to override Python's `|=` and `-=` update in-place operators, so the following could be used to insert or delete relations (or tuples):

```
r1 |= r2            #insert (union update in-place)
r1 -= r2            #delete (minus update in-place)
```

Also an update method is available which takes two `lambda` expressions: one to select which tuples to update and one to say how to update those tuples, e.g.

```
r1.update(lambda t: t.a1 == 5,
          lambda u: {'a1':0})
```

would update each tuple in `r1` that had `a1 == 5` and set `a1` to `0`. Of course, the update is just a shorthand for a delete followed by an insertion.

Updates

Access to the original tuple is also provided via the `OLD_` prefix, e.g.

```
r1.update(lambda t: t.a1 < 5,
          lambda u: {'a1':u.OLD_a1 * 10})
```

would update each tuple in `r1` that had `a1 < 5` and set `a1` to its original value multiplied by 10.

Treating Operators as Relations

We can use Python generators to provide tuples for a virtual relation.

Example: define a generator, `plus_gen`, and wrap it in a virtual relvar, `plus`:

```
def plus_gen(x=None, y=None, z=None):
    "Plus generator yielding tuples. Could just as well be called minus_gen"
    if x is not None and y is not None and z is None:
        yield {'x':x, 'y':y, 'z':x + y}
    elif x is not None and y is None and z is not None:
        yield {'x':x, 'y':z - x, 'z':z}
    elif x is None and y is not None and z is not None:
        yield {'x':z - y, 'y':y, 'z':z}
    elif x is not None and y is not None and z is not None:
        if x + y == z:
            yield {'x':x, 'y':y, 'z':z}  #i.e. True
        #else yield nothing, i.e. False
    else:
        #Note: we could go further and return tuples given just one attribute
        #      or indeed we could start yielding infinite combinations if no
        #      attributes are passed (but then non-relational?)
        raise InvalidOperation("Infinite rows")  #no pair of x,y or z

plus = View(plus_gen)
```

Treating Operators as Relations

Such a 'generated relation' can be joined as usual, e.g.

```
Relation(["x", "y"],
         [(7,   8),
          (11, 23)]) & plus
```

x	y	z
7	8	15
11	23	34

And this can then be 'composed' with arguments, e.g. what's 3 + 4?:

```
COMPOSE(GENERATE({'x':3, 'y':4}), plus)
```

z
7

and so:

```
COMPOSE(GENERATE({'x':3, 'y':4}), plus).to_tuple().z
```

would return the value `7`.

Other Implementation Details

- No plans to implement the inheritance model proposed by TTM because it overlaps so much with Python's type system. It would require a separate interpreter and virtual machine to implement it.
- Constraints are defined using `lambda` functions, where a True result satisfies the constraint, and with shorthands for key and foreign-key constraints, e.g.

```
EXAM_MARK = Relation(["StudentId", "CourseId", "Mark"],
                     [('S1', 'C1', 85),
                      ('S1', 'C2', 49),
                      ('S2', 'C1', 49),
                      ('S3', 'C3', 66),
                      ('S4', 'C1', 93),
                     ],
             {'PK': (Key, ["StudentId", "CourseId"]),
              'MarkRange': (Constraint, lambda r:
                            ALL(r, lambda t: 0 <= t.Mark <= 100))
             }
            )
```

Other Implementation Details

- Relvars are held in memory
- To improve the speed of joins, every column value is hashed so that hash-joins can be performed when using the AND function (i.e. we take the intersection of the sets of rows with matching values).

Some Outstanding Issues/Questions

- System catalog - supporting nested relations means the catalog now needs to allow recursive definitions
- Ungroup/unwrap with empty relations - need an explicit type header
- Constraint and view definitions: how best to restrict their scope and persist/reload them
- Lowercase all operators to be Pythonic (and remove methods to have a single way of doing things?)

Some Outstanding Issues/Questions

- Rename `where` method to `restrict` to match the relational operator name?
- If we define a view, what should happen if the base relvar headings subsequently change?
- Distributed persistence

Response to Initial Version

"I am impressed by the 'Third Manifesto' concept, And I am impressed by the straight forward way this concept is implemented in Python."

"I believe you've done a great work and I'm going using Dee to develop a prototype of a small application for my company"

"I saw your project to extend Python with a relational language. I think it is a great idea. I was wondering, are there any plans to continue its development? If so I'd like to make some contributions."

"For me the promise is in using the power of python classes and types, along with the ease of the python language, in a relational database setting, without all the excess of SQL or object relation mappers etc. ... I like what I've seen so far (2 hours). Hope you keep working on Dee"

"I've read the documentation and I'm really excited by the idea of giving python relational capabilities. Furthermore, as the title suggests, I'm so excited, I'd to help in any way I can."

Version 0.2

The next version of Dee includes:

- Simpler syntax for `extend` and `update` (no need to re-introduce new attributes), e.g.

```
IS_CALLED.extend(lambda t:{'Initial':t.Name[:1]})
```

  instead of:

```
IS_CALLED.extend(['Initial'], lambda t:{'Initial':t.Name[:1]})
```

- Improved internal storage (namedtuples) which will also preserve the order of heading attributes

Version 0.2

- Python 3 compatible
    - Python 3 has set literals, e.g. {1, 2, 3} so we can use this slightly better syntax, e.g.:

```
Relation(['SNO', 'SNAME', 'STATUS', 'CITY'],
         {{'SNO':'S1', 'SNAME':'Smith', 'STATUS':20, 'CITY':'London'},
          {'SNO':'S2', 'SNAME':'Jones', 'STATUS':10, 'CITY':'Paris'},
          {'SNO':'S3', 'SNAME':'Blake', 'STATUS':30, 'CITY':'Paris'},
          {'SNO':'S4', 'SNAME':'Clark', 'STATUS':20, 'CITY':'London'},
          {'SNO':'S5', 'SNAME':'Adams', 'STATUS':30, 'CITY':'Athens'},
         }
        )
```

- Persistence plug-in with driver for SQLite/PostgreSQL etc.

Version 0.2

- Comprehensive unit tests
- `IMAGE` instead of `SUMMARIZE` and `DIVIDE`, e.g.

```
S.extend(lambda t: {'TOTQD':SUM(IMAGE(SP)(['QTY']))})
```

- Simpler syntax for view definitions (no need to re-introduce heading attributes), e.g.

```
View(lambda: EXTEND(S, lambda t: {'INITIAL':t.SNAME[:1]}))
```

instead of:

```
Relation(['SNO', 'SNAME', 'STATUS', 'CITY', 'INITIAL'],
         lambda: EXTEND(S, lambda t: {'INITIAL':t.SNAME[:1]})
        )
```

Version 0.2

- Database namespaces, e.g.

```
db = Database()
...
with db: db.R1, db.R2 = A, B
```

to provide hooks for:

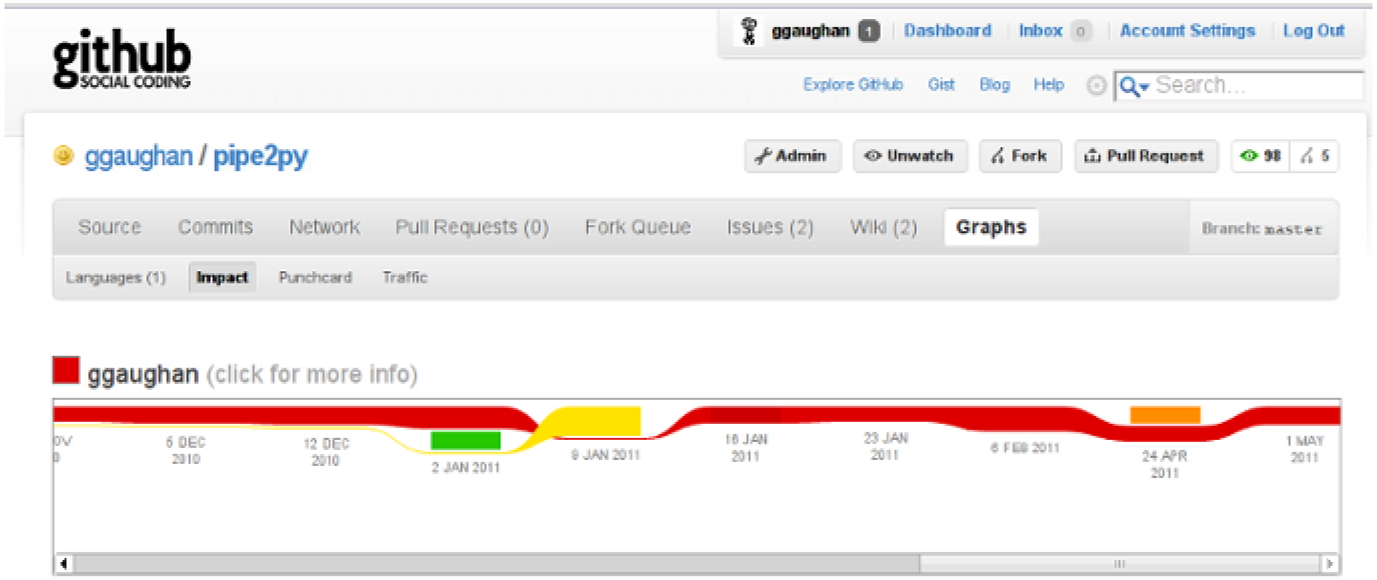
  - Persistence
  - Catalog
  - Constraint declaration & checking
  - Security declaration & checking
  - Atomic updates (which replace transactions)

Version 0.2

- Database-level constraints, e.g. (proposed syntax)

```
db = Database()
db.constraints |= {'C1': lambda:COUNT(db.r1) == 3}
```

- Will be uploaded to, and hopefully further developed on, http://github.com


github SOCIAL CODING
ggaughan 1 | Dashboard | Inbox 0 | Account Settings | Log Out
Explore GitHub Gist Blog Help Search...
ggaughan / pipe2py
Admin Unwatch Fork Pull Request 98 5
Source Commits Network Pull Requests (0) Fork Queue Issues (2) Wiki (2) Graphs Branch: master
Languages (1) Impact Punchcard Traffic
ggaughan (click for more info)
6 DEC 2010
12 DEC 2010
2 JAN 2011
9 JAN 2011
16 JAN 2011
23 JAN 2011
6 FEB 2011
24 APR 2011
1 MAY 2011

Pipelined Operations

- Relational expressions are now deferred by default (using `lambda` internally)
    - Scanning a relation yields a tuple at a time (i.e. `scan` is a Python generator)
    - More efficient memory handling for large relations (though could cache/window for internal looping)
- Potential for improving scalability - can pull from other processes
- Potential for optimisation, e.g. pass resulting pipeline to a planner for re-ordering

Future Work

- Constraint inference (may need some help translating the original algorithm!)
    - Some straightforward ones are already done, e.g. after a `rename` or `remove`
- Updates via Views (help needed!)
- System keys
- N-adic versions of some of the functions. e.g.

```
def AND(*args):
    for r in args:
        #etc.
```

Some Ideas Worth Pursuing

- A persistence plug-in for Google BigTable
- An interactive tutorial running on Google App Engine
- A 'persistence' plug-in for memcached/memcachedb (distributed memory)
- Plug Dee into the Django web framework to replace the ORM and database

Some Ideas Worth Pursuing

- Use distributed-version-control mechanisms to persist locally and merge across nodes
    - Not suitable for all applications
    - Optimised data transfers
    - Handles network outage
    - Multi-versioned, with audit
- 'Cloud' relations, e.g.

```
r1 = Relation('http://example.com/r1')
```

Summary

- 1983 - 2011 via ZX Spectrums, ThinkSQL and Dee
- Version 0.12 was well received
- Python is an excellent foundation
- Version 0.2 is under way and adds a number of improvements
- Release to github should encourage collaboration

Dee is open-source and available at www.quicksort.co.uk