Fuzzy and more. Implementing a logic calculator for comparing philosophical theories of vagueness using Structured Query Language. Part 1

Work in progress. 23.07.2018. by Marian Calborean, mc@filos.ro.

Introduction

The debate on vagueness has been greatly enriched by the development of fuzzy logic and its applications, many of which are computer-based, such as Fuzzy Markup Language - FML.

However, other philosophical theories of vagueness diverge from the basic assumptions of pluri-valuationism embedded in fuzzy logic. Moreover, inside fuzzy theory, philosophically-potent disagreements exist, e.g. the interpretation of logical connectives.

I aim to develop a tool for comparing theories of vagueness, using Structured Query Language. In order to do so, I will go through the following steps:

1. Describing the problem of vagueness;
2. Describing the relevant characteristics of SQL;
3. Defining what it is to be a theory of vagueness and other base objects in our calculator;
4. Populating the calculator with theories i.e. applying the definition above to theories of vagueness;
5. Describing the technical challenges of using SQL in this calculator;
6. The proper implementation of the calculator.

Relevant SQL snippets will be used throughout. However, the final annexes contain the full .sql database and all the scripts used for the tasks in the paper i.e. truth table generation.

This paper breaks after point 3 (to be continued), but the reader is also welcome to continue it herself.

1. Vagueness

Vagueness is a pervasive property of language, much-debated by philosophers, usually taken to include:

a. **Failure of bivalence**

   One reasonably refuses to either affirm or deny some statement such as 'John is bald'. This can be found in literature as:

   - *Borderline cases:* we can name situations where bivalence seems unwarranted
   - *Lack of sharp boundaries*: we cannot circumscribe the situations where the predicate applies (neither by definition nor by enumeration).
   - *Open texture:* seemingly well-defined terms are long-term indeterminate in face of new situations (e.g. *beer* seemed well-defined until non-alcoholic beer-like beverages were invented)

b. **A core, sui-generis characteristic of natural language and distinct from known language characteristics**

   - Not *generality* – same term applying to multiple references: e.g. *man*
   - Not *ambiguity* – one term having multiple meanings: e.g. *bank*
   - Not *failure of reference* –terms not finding a reference: e.g. *unicorn*
   - Not *indecidability*–terms whose references cannot be proven to exist or not
   - Not *lack of specificity* –terms who find their reference based on context (therefore they would possibly not find it in a different context)

c. **Soriticality (susceptibility to the heap paradox)**

   Vague terms can be employed as a starting point of the sorites paradox, the paradox of the *heap*.

   i. 1 grain is not a heap;
   ii. One grain of difference does not make heap:
   iii. Therefore, 1 million grains do not make a heap.

Or for *small*[1]:

i. 0 is small;
ii. If n is small, n + 1 is small:
iii. Therefore, every number is small.

2. Structured Query Language

SQL is a descriptive programming language based on relational algebra. We will use the MySQL 5.0 variant of the language.

a) **The relation**

The relation can be defined as the pair <C, R>, where C is a set of column names $c_1 \ldots c_n$ and R is the set of ordered pairs $<v_1..v_n>$ so that for each c_n in C there is a v_n in each subset of R.

We call each member of R a **row** and each member of C a column**.** We observe the row contains exactly one value for each column, that's why the relation can be called a **table.**

b) Stored data

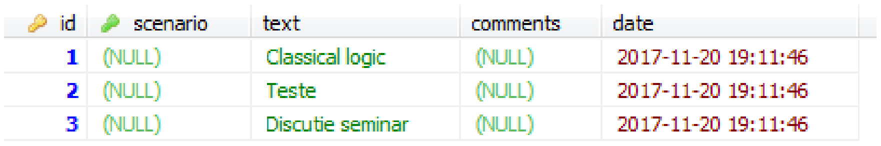
By specifying the structure of the relation (the columns), we can create a **stored table.** (Normally "table") in which we can then insert rows. It is the role of a Relational database management system software such as MySQL to retain the data and to extract it according to the specifications of the language.

c) **Statements**

In order to retrieve data as a relation, we use SELECT FROM for extracting from a stored table

```
SELECT * FROM sources
```

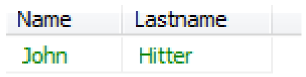
[1] Dummett 1975



id	scenario	text	comments	date
1	(NULL)	Classical logic	(NULL)	2017-11-20 19:11:46
2	(NULL)	Teste	(NULL)	2017-11-20 19:11:46
3	(NULL)	Discutie seminar	(NULL)	2017-11-20 19:11:46

We can use **SELECT** to create a relation of one row

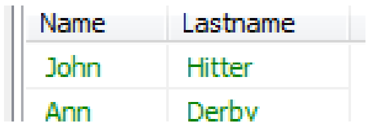
```
SELECT 'John' AS `Name`, 'Hitter' AS `Lastname`
```



Name	Lastname
John	Hitter

We use UNION to perform reunion of the R sets, provided the columns are the same:

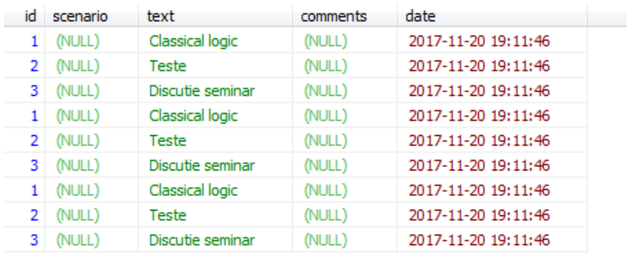
```
SELECT 'John' AS `Name`, 'Hitter' AS `Lastname`
UNION
SELECT 'Ann' AS `Name`, 'Derby' AS `Lastname`
```



Name	Lastname
John	Hitter
Ann	Derby

Of course, as a mathematical operation, reunion can be performed on stored tables repeatedly:

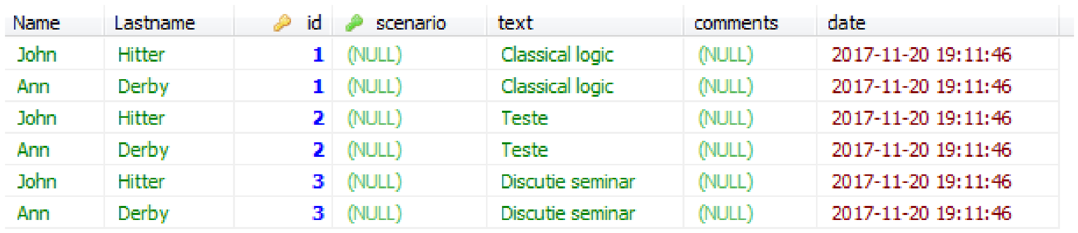
```
SELECT * FROM sources
UNION ALL
SELECT * FROM sources
UNION ALL
SELECT * FROM sources
```



id	scenario	text	comments	date
1	(NULL)	Classical logic	(NULL)	2017-11-20 19:11:46
2	(NULL)	Teste	(NULL)	2017-11-20 19:11:46
3	(NULL)	Discutie seminar	(NULL)	2017-11-20 19:11:46
1	(NULL)	Classical logic	(NULL)	2017-11-20 19:11:46
2	(NULL)	Teste	(NULL)	2017-11-20 19:11:46
3	(NULL)	Discutie seminar	(NULL)	2017-11-20 19:11:46
1	(NULL)	Classical logic	(NULL)	2017-11-20 19:11:46
2	(NULL)	Teste	(NULL)	2017-11-20 19:11:46
3	(NULL)	Discutie seminar	(NULL)	2017-11-20 19:11:46

We use JOIN to perform at the same time a reunion of C from two or more relations and the Cartesian product of their R

```
SELECT * FROM
(SELECT 'John' AS `Name`, 'Hitter' AS `Lastname`
UNION
SELECT 'Ann' AS `Name`, 'Derby' AS `Lastname`) r1
JOIN
(SELECT * FROM sources) r2
```



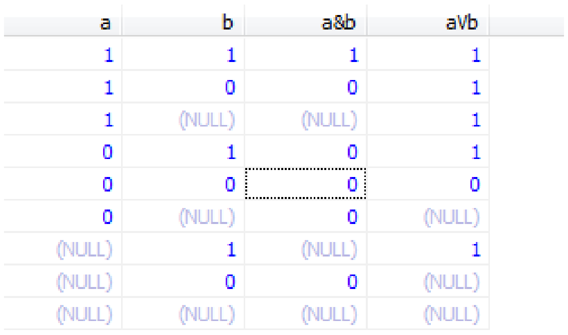
Name	Lastname	id	scenario	text	comments	date
John	Hitter	1	(NULL)	Classical logic	(NULL)	2017-11-20 19:11:46
Ann	Derby	1	(NULL)	Classical logic	(NULL)	2017-11-20 19:11:46
John	Hitter	2	(NULL)	Teste	(NULL)	2017-11-20 19:11:46
Ann	Derby	2	(NULL)	Teste	(NULL)	2017-11-20 19:11:46
John	Hitter	3	(NULL)	Discutie seminar	(NULL)	2017-11-20 19:11:46
Ann	Derby	3	(NULL)	Discutie seminar	(NULL)	2017-11-20 19:11:46

d) Some language heuristics

First, SQL implements a strong undefined value, the NULL. Any statement involving NULL except conjunctions with falsities or disjunctions with truths (or functions converting NULL to other value types) results in NULL. Also, SQL has TRUE and FALSE predicates, but they're implemented as equal to 1 respectively 0. Since SQL includes boolean operators AND and OR, it would have been possible to define some of the operators of our theories in terms of them. This is the native SQL truth table.

```
SELECT TRUE AS `a`, TRUE AS `b`, TRUE AND TRUE AS 'a&b', TRUE OR TRUE AS 'aVb'

UNION ALL

SELECT TRUE AS `a`, FALSE AS `b`, TRUE AND FALSE AS 'a&b', TRUE OR FALSE AS 'aVb'


UNION ALL

SELECT TRUE AS `a`, NULL AS `b`, TRUE AND NULL AS 'a&b', TRUE OR NULL AS 'aVb'

UNION ALL

SELECT FALSE AS `a`, TRUE AS `b`, FALSE AND TRUE AS 'a&b', FALSE OR TRUE AS 'aVb'

UNION ALL

SELECT FALSE AS `a`, FALSE AS `b`, FALSE AND FALSE AS 'a&b', FALSE OR FALSE AS 'aVb'

UNION ALL

SELECT FALSE AS `a`, NULL AS `b`, FALSE AND NULL AS 'a&b', FALSE OR NULL AS 'aVb'
```

```
UNION ALL

SELECT NULL AS `a`, TRUE AS `b`, NULL AND TRUE AS 'a&b', NULL OR TRUE AS 'aVb'

UNION ALL

SELECT NULL AS `a`, FALSE AS `b`, NULL AND FALSE AS 'a&b', NULL OR FALSE AS 'aVb'

UNION ALL

SELECT NULL AS `a`, NULL AS `b`, NULL AND NULL AS 'a&b', NULL OR NULL AS 'aVb'
```



a	b	a&b	aVb
1	1	1	1
1	0	0	1
1	(NULL)	(NULL)	1
0	1	0	1
0	0	0	0
0	(NULL)	0	(NULL)
(NULL)	1	(NULL)	1
(NULL)	0	0	(NULL)
(NULL)	(NULL)	(NULL)	(NULL)

But I prefer specifying our operators more transparently for the reader, as we shall see.

Secondly, any data of the theory stored in the data store is retrieved as data, that is a string of characters, as we saw. But it is clear from our aim that we want to *execute* the data, in order to both store and perform all operations in SQL. This is called reflection, and is implemented in SQL with a special PREPARE statement. Starting from the character string ‘SELECT 1’ store in table `imaginary` of one row at column `first`, if we want to execute it in SQL so that we get as result `1`, we need to write:

```
SET @f = (SELECT first FROM imaginary);

PREPARE stmt FROM @f;
EXECUTE stmt;
```

3. The objects (stored tables)

Each theory is about to receive a basic tri-partite characterization by accepted truth values, truth-functionality of connectives and valid inference rules (the last is work in progress).

We also model non truth-functional modifiers used in fuzzy controllers defined according to FML. There is a data store for any non-truth-functional state we might need to enable computation at those theories that might need it. Then we'll have a common dictionary of vague sentences and a way of noting the possible complex expressions in which they enter.

The final purpose is that by using recursion, the tool will derive a large number of propositions with their associated truth interpretations under each theory of vagueness and then allow their systematic comparison.

a. Theories and operators

```
CREATE TABLE `theories` (
`id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
`key` VARCHAR(8) NOT NULL,
`text` VARCHAR(255) NOT NULL,
`bool_secondary_valuation` TINYINT(1) NOT NULL DEFAULT '0',
`random_valuation_formula` TEXT NULL COMMENT 'null= cannot be',
`example_values_formula` TEXT NULL,
`comments` TEXT NULL,
`date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
UNIQUE INDEX `key` (`key`)
);
```

This creates a store for theories, characterized by unique key and ID. *bool_secondary_valuation* is a Boolean indicating whether the theory is non-truth-functional. *random_valuation_formula* will contain the SQL code generating truth-values for the theory and *example_values_formula* will contain the SQL code generating standard examples i.e. for example truth-tables.

```
CREATE TABLE `operators` (
`id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
`type` ENUM('unary','binary') NOT NULL DEFAULT 'binary',
`key` VARCHAR(8) NOT NULL,
`source` INT(10) UNSIGNED NULL DEFAULT NULL,
`source_details` TEXT NULL,
`comments` TEXT NULL,
`date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
UNIQUE INDEX `key` (`key`) USING BTREE,
INDEX `source` (`source`),
CONSTRAINT `operators_ibfk_1` FOREIGN KEY (`source`) REFERENCES `sources`
(`id`)
)
```

```
;

CREATE TABLE `theories2operators` (
`id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
`theory` VARCHAR(8) NOT NULL,
`operator` VARCHAR(8) NOT NULL,
`formula` TEXT NULL COMMENT 'will be execd as select of a left join of result
c with a and b',
`graph` POLYGON NULL DEFAULT NULL COMMENT 'one of formular/polgon needs to be
executed',
`source` INT(10) UNSIGNED NULL DEFAULT NULL,
`source_details` TEXT NULL,
`comments` TEXT NULL,
`date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
UNIQUE INDEX `theory` (`theory`, `operator`),
INDEX `operator` (`operator`),
INDEX `source` (`source`),
CONSTRAINT `theories2operators_ibfk_1` FOREIGN KEY (`theory`) REFERENCES
`theories` (`key`) ON UPDATE CASCADE,
CONSTRAINT `theories2operators_ibfk_2` FOREIGN KEY (`operator`) REFERENCES
`operators` (`key`) ON UPDATE CASCADE,
CONSTRAINT `theories2operators_ibfk_3` FOREIGN KEY (`source`) REFERENCES
`sources` (`id`) ON UPDATE CASCADE
)
```

These create a data store for operators and for the many-to-many relationships associating them with theories. The idea is that an operator can belong to more than one theory with a different formula, in order to enable comparison, that is what fields *formula* and *graph* on table *theories2operators* are about. The difference between them is that a graph can enclose the formula of the operator in a topological definition that makes it modifiable (e.g. move the graph to the left by 20%). Otherwise, formulas are procedural code, that receives inputs and outputs values, without the internal transformation being modifiable.

Finally, operators are defined by having one or two places. Operators are here not only logical connectives, but also fuzzy concepts such as *warm*, that is, it will output 0.1 for a tempreature

b. Propositions and valuations

```
CREATE TABLE `propositions` (
`id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
`operator` VARCHAR(8) NULL DEFAULT NULL COMMENT 'null=atom, !-un singur arg,
altfel fara arg->invalid',
`modifier` VARCHAR(8) NULL DEFAULT NULL,
`proposition1` INT(10) UNSIGNED NULL DEFAULT NULL,
`proposition2` INT(10) UNSIGNED NULL DEFAULT NULL,
`text` VARCHAR(255) NULL DEFAULT NULL COMMENT 'nu lipseste = atom
propozitional (fara predicat)',
`source` INT(10) UNSIGNED NULL DEFAULT NULL,
`source_details` TEXT NULL,
`source_index` VARCHAR(255) NOT NULL COMMENT 'pt cele aporpiatein text care
tb combinate',
`comments` TEXT NULL,
`date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
INDEX `source` (`source`),
INDEX `proposition1` (`proposition1`),
INDEX `proposition2` (`proposition2`),
INDEX `operator` (`operator`),
INDEX `source_index` (`source_index`),
INDEX `modifier` (`modifier`),
CONSTRAINT `propositions_ibfk_1` FOREIGN KEY (`operator`) REFERENCES
`operators` (`key`) ON UPDATE CASCADE,
CONSTRAINT `propositions_ibfk_2` FOREIGN KEY (`proposition1`) REFERENCES
`propositions` (`id`) ON UPDATE CASCADE,
CONSTRAINT `propositions_ibfk_3` FOREIGN KEY (`proposition2`) REFERENCES
`propositions` (`id`) ON UPDATE CASCADE,
CONSTRAINT `propositions_ibfk_4` FOREIGN KEY (`source`) REFERENCES `sources`
(`id`) ON UPDATE CASCADE,
CONSTRAINT `propositions_ibfk_5` FOREIGN KEY (`modifier`) REFERENCES
`modifiers` (`key`) ON UPDATE CASCADE
)


CREATE TABLE `valuations` (
`id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
`theory` VARCHAR(8) NOT NULL,
`proposition` INT(10) UNSIGNED NOT NULL,
`secondary` INT(10) UNSIGNED NULL DEFAULT NULL COMMENT 'theory dependent,
null = fallback in those that accept it non-null',
`val` DECIMAL(8,4) NULL DEFAULT '0.0000' COMMENT 'null=undef at three
valued',
`source` INT(10) UNSIGNED NULL DEFAULT NULL COMMENT 'null = base value',
`source_details` TEXT NULL,
`comments` TEXT NULL,
`date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
UNIQUE INDEX `theory` (`theory`, `proposition`, `secondary`),
INDEX `proposition` (`proposition`),
INDEX `source` (`source`),
INDEX `secondary` (`secondary`),
CONSTRAINT `valuations_ibfk_1` FOREIGN KEY (`theory`) REFERENCES `theories`
(`key`) ON UPDATE CASCADE,
CONSTRAINT `valuations_ibfk_2` FOREIGN KEY (`proposition`) REFERENCES
`propositions` (`id`) ON UPDATE CASCADE,
CONSTRAINT `valuations_ibfk_3` FOREIGN KEY (`source`) REFERENCES `sources`
(`id`) ON UPDATE CASCADE
)
;
```

The proposition store can hold either atomic sentences or complex sentences by indicating the operator used and the two subcomponents (their computation will be done automatically, based on the properties of the Polish notation). This is a recursive data store i.e. new propositions formed will be added with fields *proposition1* and *proposition2* containing the ID's of other propositions already in the data store.

The valuations are the association of theories with propositions, either by initial stipulation (if the proposition is atomic) or according to the rules of the proposition operator applied according to its unique relationship with the theory. Field *theory* holds the theory ID. Field *proposition* holds the proposition ID. Field *val* holds the valuation (NULL = the undefined value for such theories as three-valued interpretations)

c. Modifiers

```
CREATE TABLE `modifiers` (
`id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
`key` VARCHAR(8) NOT NULL,
`source` INT(10) UNSIGNED NULL DEFAULT NULL,
`source_details` TEXT NULL,
`comments` TEXT NULL,
`date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
UNIQUE INDEX `key` (`key`) USING BTREE,
INDEX `source` (`source`)
)
;

CREATE TABLE `theories2modifiers` (
`id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
`theory` VARCHAR(8) NOT NULL,
`modifier` VARCHAR(8) NOT NULL,
`graph_formula` TEXT NOT NULL COMMENT 'takes @graph as parameter, outputs
another graph',
`source` INT(10) UNSIGNED NULL DEFAULT NULL,
`source_details` TEXT NULL,
`comments` TEXT NULL,
`date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
UNIQUE INDEX `theory` (`theory`, `modifier`),
INDEX `operator` (`modifier`),
INDEX `source` (`source`),
CONSTRAINT `theories2modifiers_ibfk_1` FOREIGN KEY (`theory`) REFERENCES
`theories` (`key`) ON UPDATE CASCADE,
CONSTRAINT `theories2modifiers_ibfk_2` FOREIGN KEY (`modifier`) REFERENCES
`modifiers` (`key`) ON UPDATE CASCADE,
CONSTRAINT `theories2modifiers_ibfk_3` FOREIGN KEY (`source`) REFERENCES
`sources` (`id`) ON UPDATE CASCADE
)
```

```
;
```

Modifiers are non-truth functional operators that modify operators (only those that implement a graph). These allow us to implement the specifications of FML and later create fuzzy controllers for the fuzzy theory in the calculator. Supposing we have operator `hot` defined as above by a graph so that its value starts at 0.01 when temperature (that is, the initial valuation) is 24 degrees and grows linearly to 1 when temperature is 26 degrees (then stays at 1). Then ‘NOT TOO` can be defined as a modifier that flips the operator graph horizontally, so that it starts at 1 at low temperatures, begins descreasing at about 24 degrees and reaches 0 when temperature is 26 degrees.

Modifiers are theory-specific, having many-to-many relationships with the theories. Field *graph_formula* is the one that would do the work, as commented.

d. Sources

```
CREATE TABLE `sources` (
`id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
`scenario` VARCHAR(255) NULL DEFAULT NULL COMMENT 'scenario
number/description',
`text` TEXT NOT NULL,
`comments` TEXT NULL,
`date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
INDEX `scenario` (`scenario`)
)
COLLATE='utf8_general_ci'
ENGINE=InnoDB
AUTO_INCREMENT=4
;
```

Sources is a data store just for the textual references of our objects. For example, we will create a source for Edgington’s article `Vagueness by Degrees` with which we will associate the corresponding theory, operator, example propositions etc.

[to be continued]