J Inf Process Syst, Vol.19, No.1, pp.1~16, February 2023
https://doi.org/10.3745/JIPS.04.0262

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Enhanced Regular Expression as a DGL for Generation of Synthetic Big Data

Kai Cheng* and Keisuke Abe

Abstract

Synthetic data generation is generally used in performance evaluation and function tests in data-intensive applications, as well as in various areas of data analytics, such as privacy-preserving data publishing (PPDP) and statistical disclosure limit/control. A significant amount of research has been conducted on tools and languages for data generation. However, existing tools and languages have been developed for specific purposes and are unsuitable for other domains. In this article, we propose a regular expression-based data generation language (DGL) for flexible big data generation. To achieve a general-purpose and powerful DGL, we enhanced the standard regular expressions to support the data domain, type/format inference, sequence and random generation, probability distributions, and resource reference. To efficiently implement the proposed language, we propose caching techniques for both the intermediate and database queries. We evaluated the proposed improvement experimentally.

Keywords

Big Data Analytics, Data Generation Language (DGL), Performance Analysis, Regular Expression, Synthetic Data Generation, Type/format Inference

1. Introduction

Synthetic data are widely used in various areas of research, such as data analytics and artificial intelligence. Synthetic data are vital for the empirical evaluation of algorithms and methodologies in big data analytics. Synthetic datasets are used in the benchmarking of data-intensive applications, where tools for populating a database with a large volume of artificial data satisfy database constraints and statistical distributions [1-5]. The use of synthetic data is a common practice for stock market data analytics, health-care, energy analytics, image recognition, and workload prediction [6,7]. Such use has also been studied for different purposes such as privacy-preserving data publishing (PPDP) [8] or statistical disclosure limit/control (SDL/SDC) [9]. This is important for a better understanding of machine learning algorithms and methodologies that require large-scale experimentation.

There have been a number of previous studies conducted on automated data generation [10-12]. However, such studies have been limited by the types of data that can be generated, and the text data obtained are often meaningless random strings. Furthermore, tools or languages have been developed for specific tasks, and thus they are generally unsuitable for other application domains.

Manuscript received January 15, 2021; first revision November 29, 2021; accepted December 26, 2021.
* **Corresponding Author:** Kai Cheng (chengk@is.kyusan-u.ac.jp)
Dept. of Information Science, Kyushu Sangyo University, Fukuoka, Japan (chengk@is.kyusan-u.ac.jp, abe@is.kyusan-u.ac.jp)

In this study, we propose the use of regular expressions as a data generation language (DGL) for synthetic data generation. Given a regular expression, a set of strings that exactly matches it is generated. For example, regexp /090-\d{4}-\d{4}/ specifies the typical pattern of mobile phone numbers in Japan. The numbers 090-1234-5678 and 090-2345-5432 are instances of this pattern. However, standard regular expressions are insufficiently expressive as a data generation language. In this study, we introduce important extensions, such as sequential numbers, random numbers, and random dictionary sampling, that make regular expressions a powerful tool for generating realistic datasets with content and probability distributions that match the target database. Our contributions include the following: (1) a regular expression-based general-purpose DGL for synthetic big data generation, (2) important enhancements including data domains, type/format inference, and resource references, and (3) support for various probability distributions and resource definitions.

The remainder of this article is organized as follows. After summarizing related studies in Section 2, we introduce preliminaries on regular expressions in Section 3. In Section 4, we describe the main enhancements to the standard regular expressions. Following the case study in Section 5, we introduce implementation and performance issues. Finally, we summarize the results and provide some concluding remarks.

2. Related Work

Because real datasets are often subject to privacy regulations and might not have typical characteristics, synthetic data are a necessary solution. Because synthetic data are available without legal, ethical, or commercial restrictions, researchers can conduct controlled experiments, focusing on changes with respect to specific characteristics while leaving other aspects untouched. This makes it easier to conduct a qualitative assessment on whether a learning algorithm can achieve its design goal with respect to the ground truth in synthetized datasets [13-16].

Data generators are the heart of database system analysis and have been discussed in both industry and academia [17-20]. In [21], the authors explored different techniques for quickly generating billion-record synthetic databases for TPC-A. Stephens and Poess [22] presented MUDD, a multidimensional data generator. Originally designed for TPC-DS, MUDD can generate up to 100 terabytes of flat file data in hours by utilizing modern multiprocessor architectures, including clusters. Its novel design separates data generation algorithms from data distribution definitions.

Bruno and Chaudhuri [23] introduced DGL, a simple specification language used to generate databases with complex synthetic distributions and inter-table correlations. Although their study provided support for different data types, the definition of a text pattern and how such patterns are organized into a dataset are quite different from our usage. In addition, our proposed method is more general and flexible for synthetic data generation.

3. Preliminaries

In this section, we review regular expressions and their probabilistic variants. We show that regular expressions are concise and powerful when used in a pattern definition language. However, as a DGL,

the lack of support for data domains is a critical drawback.

3.1 Regular Expressions

A regular expression (regex for short) is a sequence of characters that defines patterns in text [24-26]. A pattern consists of one or more characters, operators, or constructs. Each character in a regular expression is either a meta-character with a special meaning or a regular character with a literal meaning.

Character set []. By placing the characters to be matched between square brackets, such as in [aeiou], a character set is defined such that any character in it can be matched. The regex engine matches only one of the characters in a character set. A hyphen inside a character set specifies a range of characters, for instance, [a-z], [0-9a-fA-F], and [SA-E]. Here, \d, and \w are shorthand for the character sets [0-9] and [0-9a-zA-Z_].

Quantifiers ?, *, +, {n, m}. A quantifier defines the number of times a preceding token is repeated. An asterisk (*) means that the preceding token is repeated zero or more times. The question mark (?) tells the regex engine to optionally match the preceding token. The plus operator (+) tells the regex engine to repeat the token one or more times. Here, $\{n, m\}$ is an additional quantifier that specifies the number of times a token can be repeated. The value of n is zero or a positive integer number, indicating the minimum number of matches, and m is an integer equal to or greater than n indicating the maximum number of matches.

Alternation |. The alternation operator has the lowest precedence among all regression operators. The regex engine will match everything to the left of an alternation operator or everything to its right.

Capturing group (). A regular expression can be grouped by placing a part of it inside parentheses. This allows the application of a quantifier to the entire group or restricts an alternation to a part of the regex. In addition to grouping part of a regular expression together, parentheses also create a numbered capturing group. A portion of the string matched by a part of the regular expression is stored inside the parentheses. Capturing groups make it easy to extract a part of a regex match. The numbered capturing groups can be reused later inside the regular expression via back references.

Backreference, \1–\9. Backreferences match the same text as previously matched by a capturing group. By placing the opening tag into a group, the captured tag name can be reused by a backreference to match the closing tag. For example, in <(h[1-6])>.*?</\1>, captures one of the headline tags and references them later when matching the closing tag.

3.2 Probabilistic Regular Expressions

Probabilistic regular expressions are succinct representations of a corresponding probabilistic regular language. Probabilistic regular grammar is regular grammar with the probability of production rules. A probabilistic grammar G can be defined by a quintuple G = (N, T, R, S, P), where N is the set of non-terminal symbols, T is the set of terminal symbols, R is the set of production rules, S is the start symbol, and P is the set of probabilities of production rules. Consider a grammar with N = {E, A}, T = {a, b, c}, R = $\{E \rightarrow aE, E \rightarrow bE, E \rightarrow bA, A \rightarrow aA, A \rightarrow c\}$ and S = {E}, P = {(0.50, 0.25, 0.25), (0.40, 0.60)}. The production rules for non-terminal E are as follows:

$$E \to aE, \quad p = 0.50$$
$$E \to bE, \quad p = 0.25$$
$$E \to bA, \quad p = 0.25$$

The language defined by this grammar is a set of strings with at least two *c* endings, such as *abcc* and *aabcc*. With the specified production probabilities, a string *aaabcc* of length 6 is more likely to occur than *bbbbcc* of the same length. By contrast, in classic regular grammar, uniform distributions are always assumed for rules with the same left-hand nonterminal. We extend the classic regular expressions to a probabilistic version by introducing a *probabilistic alternation*:

$$E = E_1 : p_1 \mid E_2 : p_2 \mid \cdots \mid E_n : p_n$$

In (1), E_i $(i = 1,2,\ldots,n)$ is either a classic or probabilistic regular expression, and p_i is the probability of E_i. The following is an example defining the grades for a college class (S = excellent, A = good, B = well done, C = pass, and D = do not pass).

$$[\mathrm{S}] : 0.10 \mid [\mathrm{A} - \mathrm{C}] : 0.70 \mid [\mathrm{D}] : 0.20$$

A, B, and C are the most common grades, whereas S and D are less common, and ABC is therefore chosen uniformly at random.

4. Enhanced Regex as a DGL

Although regular expressions are a simple language for specifying patterns in text, their context-free characteristics limit their expressiveness. For example, a regular expression cannot naturally define valid data types, such as dates. A regular expression has difficulty validating months in [1…12] and days in [1…31]. In this section, we introduce data domains and other important enhancements to make regex a powerful data generation language.

4.1 Data Domains

Synthetic datasets have many attributes, each of which has a data domain. A data domain is the collection of values permitted for an attribute. It is important to provide a suitable domain for synthesizing meaningful data for each attribute. The standard domain types in most database management systems (DBMSs) include texts, numerals, dates, and times. These primitive data domains are insufficient for data synthesis because the data values in a real dataset only contain meaningful data that are usually a subset of a data domain. For example, a reasonable domain for a name attribute can be a variable character, such as VARCHAR(16), and thus any string no longer than 16 characters in length is valid. However, in a real database, only a small subset of meaningful names is considered valid.

In this section, we describe the domain definition mechanism. Basically, we provide the types of domain definitions, i.e., *set* and *range*. A set is a finite collection of values in memory or external storage, such as files or databases. A range is a sequence of values in which the next value can be obtained by computing the previous one.

4.1.1 Set domains

A set domain is defined as a set in the following form:

$$set(resource, case = None) \tag{1}$$

The resource parameter specifies where to obtain the data. A resource can be a set of values in memory or values from a file or database. The *case* option defines whether or how to convert values in a list. The case can be lower, upper, or title case. Now, let us explain how resources are organized and utilized in the domain definition.

In-memory resources: In most programming languages, in-memory resources are a list or array. The following list of fruit names is an example of an in-memory resource:

$$fruit = ['apple',\ 'banana',\ 'cherry',\ 'pear'$$
$$grade = ['S','A','B','C','D']$$

File-based resources: Resources are organized into files with a tree-like directory structure (Fig. 1). All files and directories are under a common root $dict$ and their paths are specified in a dot-separated form, that is, $root.parent.child$. The $child$ part in the path can be either a file (leaf) or directory (non-leaf). If it is a file, the content of the file (item per line) is read and parsed into a list. If it is a directory, all files under the directory, and recursively the files in the subdirectories, are combined into a single resource.

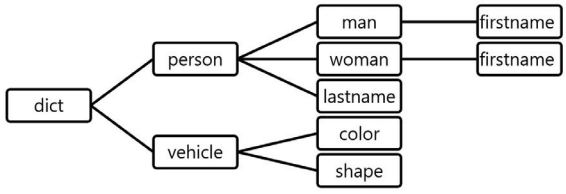

Fig. 1. Directory structure of file-based resources.

Wildcards * can be used for abbreviated path specifications such $root.parent.*.descendant$, and can be shortened as $root.parent.descendant$ if unambiguous (if $descendant$ is not a real directory under $parent$). For example, $dict.person.firstname$ refers to the first name of a person, both men and women, which is a shorthand of $dict.person.*.firstname$. To use last names, and women's first names, we can specify the resources as follows:

$lastname = dict.person.lastname$
= ['*Stewart*', '*Morgan*', '*Trump*', '*Bush*', '*Scott*', '*Abe*', …]

$woman = dict.person.woman.firstname$
= ['*Susan*', '*Sophia*', '*Jessica*', '*Jennifer*', '*Anne*', …]

The files are in *dict/person/lastname* and *dict/person/woman/firstname*, where *lastname* and *firstname* are the target resource files. We can use abbreviated path specifications to name the resources.

$firstname = dict.person.firstname$
= ['*Douglas*', '*James*', '*Oliver*', '*Benjamin*', ..., '*Susan*', '*Sophia*', '*Jessica*', '*Jennifer*']

All first names under *dict/person* form a single resource, including *dict/person/man/firstname* and *dict/person/woman/firstname*.

DB-based resources: Resources are also supported by backend databases. Suppose there is a table called fruits in the database, containing columns such as name and color. A resource can be defined through SQL:

$$fruitcolor = db('\textbf{select}\ name, color\ \textbf{from}\ fruits\ \textbf{order by}\ name')$$
$$= [['apple', 'red'], ['banana', 'yellow'], ['cherry', 'pink']]$$

4.1.2 Sequence domains

Many data domains can be treated as a sequence of consecutive values defined for certain data types. Such domains can be defined as (2). A sequence is built from a starting point, and the next values can be obtained by applying a delta to the previous value.

$$range\ (start, stop = infinity, delta, format) \tag{2}$$

Delta is the difference between the consecutive values. Each delta value has a number and optional unit in the form of $< delta >:=< number > [< unit >]$. When more than one delta value is given, it is repeatedly applied individually until reaching the stop point. A negative delta value is allowed, which means a decrement of the value. For example,

delta = [*5*]: increment of 5.
delta = [*5, 2*]: increment of 5, followed by increment of 2
delta = [*-0.2*]: decrement of 0.2

Note that because no data type is explicitly given in (2), it must be determined implicitly. Alternatives to (2) can be used when the data type is determined.

$integer\ (start, stop = infinity, delta, format)$
$decmial\ (start, stop = infinity, delta, format)$
$datetime\ (start, stop = infinity, delta, format)$

A *unit* is associated with the type of data time that specifies the time unit for a delta value. A unit can be D for day, M for month, Y for year, W for week, h for hour, m for minute, and s for second. For example,

delta = ['*2D*']: increment of 2 days
delta = ['*W*']: increment of 1 week, or equal to 7 days (7D)
delta = ['*-3h*', '*30m*']: initial decrement of 3 h, followed by increment of 30 min.

Finally, a domain can have a format definition when formatted values are required, for example, precision for a float number or leading zeros for integers. In the following section, we provide some examples of sequence domains.

Integer domains

$\boldsymbol{range}(1) = [1, 2, 3, \cdots, \infty]$
$\boldsymbol{range}(1, 120) = [1, 2, 3, \cdots, 119]$
$\boldsymbol{range}(001, 120) = [001, 002, 003, \cdots, 119]$

$\boldsymbol{range}(1, 120, \boldsymbol{format} = '\ 03d') = [001, 002, 003, \cdots, 119]$
$\boldsymbol{range}(1, \boldsymbol{delta} = [3]) = [1, 4, 7, 10, \cdots, \infty]$
$\boldsymbol{range}(1, 120, \boldsymbol{delta} = [3, 1]) = [1, 4, 5, 8, 9, \cdots, 116, 117]$

Decimal domains

$\boldsymbol{range}(0, 5, \boldsymbol{delta} = [0.01]) = [0, 0.01, 0.02, \cdots, 4.49]$
$\boldsymbol{range}(0.00, 5.00, \boldsymbol{delta} = ['0.01']) = [0.00, 0.01, 0.02, \cdots, 4.49]$
$\boldsymbol{range}(0, 5, \boldsymbol{delta} = [0.01], \boldsymbol{format} = '.2f') = [0.00, 0.01, 0.02, \cdots, 4.49]$
$\boldsymbol{range}(2, -2, \boldsymbol{delta} = [-0.1]) = [2, 1.9, 1.8, \cdots, -1.8, -1.9]$
$\boldsymbol{range}(2, -2, \boldsymbol{delta} = [-0.1], \boldsymbol{format} = '.2f') = [2.00, 1.90, 1.80, \cdots, -1.80, -1.90]$

Datetime domains

$\boldsymbol{range}('2020\text{-}4', '2021\text{-}3', \boldsymbol{delta} = ['W']) = ['2020\text{-}4\text{-}1', '2020\text{-}4\text{-}8', \cdots]$
$\boldsymbol{range}('9{:}00', '19{:}00', \boldsymbol{delta} = ['100m']) = ['9{:}00', '10{:}40', '12{:}20', '14{:}00', \cdots]$
$\boldsymbol{range}('9{:}00', '19{:}00', \boldsymbol{delta} = ['100m', '20m']) = ['9{:}00', '10{:}40', '11{:}00', '12{:}40', \cdots]$

4.1.3 Probability distributions in a domain

Data synthesis is based on a random sampling of domains. Uniform sampling is not always suitable under all situations. Some skews often exist in real datasets. A domain can be associated with a probability distribution. A distribution can be given as a list of categorical domains as follows:

$$p = \{v_1 : p_1, v_2 : p_2, \cdots, v_m : p_m\}, \quad \left(\sum_{i=1}^{m} p_i \leq 1\right) \tag{3}$$

or as a function $p = f(v)$ for a numerical domain. For an example $p = \{'apple' : 0.3, 'banana' : 0.45\}$ is a probability distribution in the domain $['apple', 'banana', 'cherry', 'pear']$, where "apple" has a smaller probability than "banana" but a larger one than "cherry" and "pear" because the two share a probability of 0.25. For another example, $p = \{'Stewart' : 0.2, 'James' : 0.35\}$ indicates that "Stewart" and "James" are the most popular male first names in the domain.

By default, instances are uniformly extracted at random from the domain. Alternatively, the generation process can follow a statistical distribution. The distributions supported by DGL are listed in Table 1. In Section 4.2.2, we provide a formula for supporting this feature.

Table 1. Probability distribution support

Distribution	Description
Uniform [Default]	A uniform distribution is a continuous probability distribution and is concerned with events that are equally likely to occur.
Normal	A normal distribution, also known as a Gaussian distribution, is a probability distribution that is symmetric about the mean, showing that data near the mean are more frequent in occurrence than data far from the mean.
Exponential	An exponential distribution is the probability distribution of the time between events in a Poisson point process, in which events occur continuously and independently at a constant average rate.
Poisson	A Poisson distribution is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time or space if these events occur with a known constant mean rate and independently of the time since the last event.
Zip	Zip is a distribution of probabilities of occurrence following Zipf's law.

4.2 Primitive Generators

A generator is an iterator in a domain that traverses the domain and returns a set of samples from the domain. There are two types of primitive generators: sequence generator and random generator.

4.2.1 Sequence generator $domain.seq()$

$domain.seq(size)$: A sequence generator returns each value while traversing a domain. Function $seq()$ has a parameter size that specifies the sample size.

4.2.2 Random generator $domain.rng()$

$domain.rng(size, replace = True)$: A random generator returns a random sample from the domain. By default, the samples were uniformly extracted at random. There are two parameters, i.e., one for the sample size and the other for sampling with/without a replacement.

In addition to a uniform generator, other generators for typical probability distributions are also supported.

$$domain.poisson(size, replace = True)$$
$$domain.normal(size, replace = True)$$
$$domain.zipf(size, replace = True)$$
$$domain.exponential(size, replace = True)$$

We explain the use of generators with different data domains. First, we define some domains (without *a set* wrapper for brevity).

grade=['S','A','B','C','D']
firstname = dict.person.firtname
lastname = dict.person.lastname
classtime = *range* ('9:00','19:00', delta='time[100m]')

When we call a generator *grade*.rng(3), a list of random samples ['B', 'A', 'S'] is returned. If we call *firstname.rng*(4), ['Douglas', 'Susan', 'Anne', 'Jennifer'] will be output.

A call of *classtime.seq*() will return ['9:00', '10:40', '12:20', '14:00', '15:40', '17:20'] as class times. In addition, *classtime.rng*(replace=False) returns ['17:20', '10:40', '15:40', '9:00', '12:20', '14:00'] as a random sample. When *classtime.normal*(replace = True) is called, we can obtain a random sample with a replacement ['14:00, '12:30', '15:40', '9:00', '12:20', '14:00'], which follows a normal distribution.

4.3 Regular Expressions as a DGL

We now present a regular expression-based DGL. Some features in regex, such as anchor, ^$, and assertion, are primarily provided for pattern-matching or replacement. We exclude these features and select the following features or operations for data generation:

(1) Literal, e.g., abc, k20rs121,
(2) Character set or character class, e.g., [a-z]
(3) Shorthand character class, e.g., \d, \s, \w
(4) Negated character set, e.g., [^aeiou]
(5) Concatenation
(6) Alternation/union |

(7) Quantifiers ?*+{n}{n,m}
(8) Grouping ()
(9) Backreference, e.g., \1, \2, ..., \9

4.3.1 Resource reference

To support data domains in a regular expression, we introduce a new feature called a *resource reference* for the use of resources outside a regular expression. Here, resources can be another regular expression or a domain and an associated generator. There are two types of resource references: n*amed reference* and *index reference*.

$$\%\{name\},\ \ \%0\sim\%9 \tag{4}$$

A named reference takes the form *%{name}*, and *name* refers to a named resource. The index reference uses a single digit (0–9) to specify the target resource. As a difference between them, the index reference is used when resources are provided as a list instead of individually.

Suppose we have the following provided resources.

man = dict.person.man.firstname
woman = dict.person.woman.firstname
lastname = dict.person.lastname
birthday = *range*('1980-1','2000-12', delta=['D'])

E = r/(M, %{*man*}:0.3 | F, %{*woman*}:0.7) %{*lastname*}, %{*birthday*}, 0[7-9]0-\d{4}-\d{4}/ (5)

A regular expression with resource references is written as (5), which describes a pattern with a gender of M or F, followed by the first name (30% men and 70% women) and the last name. A birthday is a randomly chosen date from 1980-1-1 to 2020-12-31, and a cell phone number begins with 070, 080, or 090, followed by two 4-digit numbers.

4.3.2 Regex-based generator $reg()$

$\boldsymbol{reg}(\boldsymbol{E}, size, resources)$: A regex-based generator, short for *reg*, is an iterator that returns a set of strings that exactly match the given regex $\boldsymbol{E}$. The size parameter specifies the sample size. The resources parameter provides a list of resources. Each resource can be another *reg* or set/sequence generator in a certain domain.

Considering a *reg* for the regex in (5), we can define a generator with a sample size of 7 as follows:

$$persons\ =\ reg(E, size = 7) \tag{5}$$

When calling the generator, the output will be as follows:

gender	name	birthday	phone number
M,	Douglas Mitchell,	1982-4-3,	070-1234-5678
F,	Jennifer Stewart,	1990-3-14,	090-2341-5432
M,	Ernest Morgan,	1984-7-4	080-2145-5214
M,	James Scott,	1999-1-15,	070-1335-2125
F,	Jessica Simmons,	1988-6-8,	080-2748-5732

Skew distributions can be generated by applying the probability distributions. It is important to achieve an inter-table dependency. Consider the following example:

p = {'apple': 0.3, 'banana': 0.45}
fruits = db('select name from fruits').*rng(p)*
person = db('select id as person_id from person')
date = *range*('2020-1', '2020-12', delta=['D'])
qty = *range*(100,500, delta=[10])

$$\mathbf{E} = \text{r/\%0,\%\{}person\text{\},\%\{}fruits\text{\},\%\{}date\text{\},\%\{}qty\text{\},\%1/} \quad (6)$$

There were two index references, %0 and %1. These are provided in the list of resources. Here, %0 is for id and %1 is for the status regarding whether the person ran.

$$persons = reg(E, size = 4, resources = [range(01,10), r/[YN]/])$$

When calling the generator, the output will be as follows:

id	person_id	fruit	date	qty (g)	running or not
01,	12,	'apple',	'2020-3-12',	150,	Y
02,	11,	'banana',	'2020-5-9',	230,	N
03,	18,	'apple',	'2020-8-25',	170,	N
04,	14,	'pear',	'2020-6-1',	300,	Y

5. Case Study

In this section, we describe the application of the proposed DGL to the TPC-H benchmark [27]. The TPC-H benchmark was designed using TPC for decision-support systems. Commercial database vendors and related hardware vendors use these benchmarks to demonstrate the superiority and competitive edge of their products. TPC-H consists of separate and individual tables (base tables) and the relationships between columns in these tables. The data types in TPC-H include identifiers, integers, [big] decimals, fixed text, variable text, and dates, all of which can be easily translated into DGL data types.

5.1 Using Data Domains in TPC-H

TPC-H uses several dictionaries to generate meaningful text strings. For example, P_TYPE is a combination of words from three sets.

PartSize = [‘Standard’, ‘Small’, ‘Medium’, ‘Large’, ‘Economy’, ‘Promo’},
PartCoat = [‘Anodized’, ‘Burnished’, ‘Plated’, ‘Polished’, ‘Brushed’}
PartMaterial = {‘Tin’, ‘Nickel’, ‘Brass’, ‘Steel’, ‘Copper’}

The following regular expression defines values for P_TYPE:

size = dict.tpc-h.PartSize
coat = dict.tpc-hPartCoat
material = dict.tpc-h.ParMaterial
E = *r*/%{*size*}\s%{*cost*}\s%{*material*}/

The output of reg(E, size=20) will be a list of strings such as

[‘Small Plated Brass’, ‘Economy Burnished Copper, …].

Here, P_CONTAINER is generated by concatenating syllables selected at random from each of the two lists and separated by a single space.

ContainerSize = [‘SM’, ‘Med’, ‘Jumbo’, ‘Wrap’]
ContainerType = [‘Case’, ‘Box’, ’Jar’, ‘Pkg’, ‘Pack’, ‘Can’, ‘Drum’]

The following regular expression defines values for P_CONTAINER:

size = dict.tpc-h.ContainerSize
type = dict.tpc-h.ContainerType
E = *r/%{size}\s%{type}/*

The output of *reg*(E, size=20) will be a list of strings such as [‘Med Can’, ‘Wrap Jar’, …].

5.2 Populating Tables in TPC-H

It is extremely easy to define single values for individual columns in TPC-H tables. Using resource references in the proposed DGL, it is also possible to generate inter-table dependencies between columns with respect to foreign keys. In this section, we present the table definition with DGL expressions for each column and the relationship between tables.

5.2.1 PART table (SF: Scaling Factor)

P_PARTKEY: identifier, => *range(1, SF*200000).seq()*
P_NAME: variable text, size 55, => *r/\w{10,55}/*
P_MFGR: fixed text, size 25, => *r/\w{25}/*
P_BRAND: fixed text, size 10, => *r/\w{10}/*
P_TYPE: variable text, size 25, => *r/%{size}\s%{cost}\s%{material}/*
P_SIZE: integer, => *range(1,100).rng()*
P_CONTAINER: variable text, size 10, => *r/%{size}\s%{type}/*
P_RETAILPRICE: decimal, => *range(0.00, 10000.00).rng()*
P_COMMENT: variable text, size 23, => *r/\w{1,23}/*
Primary Key: P_PARTKEY

5.2.2 SUPPLIER table

S_SUPPKEY: identifier, => *range(1, SF*10000).seq()*
S_NAME: fixed text, size 25, => *r/\w{25}/*
S_ADDRESS: variable text, size 40, => *r/\w{10,40}/*
S_NATIONKEY: Foreign Key to N_NATIONKEY,
=> *db(‘select N_NATIONKEY from N_NATION’)*
S_PHONE: fixed text, size 15, => *r/%0-%1-%2-%0/*
resources=[range(1,999).rng(), range(100,999).rng(), range(1000,9999).rng()]
S_ACCTBAL: decimal, => *range(100.00, 100000.00).rng()*
S_COMMENT: variable text, size 101, => *r/\w{1,101}/*
Primary Key: S_SUPPKEY

5.2.3 PARTSUPP table

PS_PARTKEY: Identifier Foreign Key to P_PARTKEY
=> db('select P_PARTKEY from P_PART')
P**S**_*SUPPKEY*: Identifier Foreign Key to S_SUPPKEY
=> db('select P_SUPPLYKEY from P_SUPPLY')
PS_AVAILQTY: integer, *=> range(1,10000).rng()*
PS_SUPPLYCOST: decimal, *=> range(0.00, 100000.00).rng()*
PS_COMMENT: variable text, size 199 , *=> r/\w{1,199}/*
Primary Key: **PS**_*PARTKEY*, **PS_SUPPKEY**

The skewness of the data distribution is an important factor in evaluating the join performance. As pointed out in [28-30], although uniform data distributions are a design choice for the TPC-H benchmark, it has been universally recognized that a data skew is prevalent in data warehousing. The following example demonstrates the capability of the proposed DGL to generate skewed distributions.

part = db('select P_PARTKEY from P_PART')
supply = db('select P_SUPPLYKEY from P_SUPPLY')
E = *r/(%{part}:0.35,%{supply}:0.65),%0,%1, \w{1,199}/*

This example describes the probabilistic correlation between foreign keys, with 35% from PART and 65% from SUPPLIER.

6. Implementation and Performance Evaluation

To implement the proposed DGL for data generation, the system comprises the following components: Resource Manager, RegExp Parser, Random Generator, and Instance Generator.

The Resource Manager supports file- and database-based resource management. It wraps external resources and exports them to the primitive generators. A RegExp Parser accepts the DGL expressions as input, parses the expressions, and builds the corresponding generators for the subpatterns [31,32]. A type/format inference module assists the parser in inferring the data type and formats implied by the literature. A Random Generator is used to generate random numbers that transform them into appropriate types and formats. Based on the Random Generator, the other generator extracts a sample from the given data domains. Finally, an Instance Generator comprises instances of the subpatterns and outputs the results.

To improve the efficiency of the data generation, caching technology can be effectively used: partial regular expression caching and DB caching can be expected to improve the efficiency of the data generation.

(1) RegExp caching: To reduce the cost of parsing (compiling) regular expressions, caching the internal form once compiled is more time efficient. Each regular expression can be associated with a compiled form, thus reducing the cost of the compilation process.

(2) DB caching: The cost of dictionary sampling, that is, sample extraction from user-defined character classes, can be reduced by DB caching. Although standard DBMS caching technology can be used, it is

necessary to shuffle the cache data such that the same result is not obtained each time the cache is used.

Experiments were conducted to evaluate the efficiency of the two caches. The evaluation targets are listed in Table 2. The change in execution time was examined depending on whether the regular expression analysis result cache (RegExp Caching) or the database cache (DB Caching) was used. Fig. 2 shows the experiment results. A regular expression analysis was conducted using the PHP package ReverseRegex [33]. The execution time was calculated as the average of 10 repetitions. It was shown that DB caching is more efficient than RegExp caching in reducing the execution time.

Table 2. Experiment settings

Name	RegExp caching	DB caching
TT	Yes	Yes
FT	No	Yes
TF	Yes	No
FF	No	No

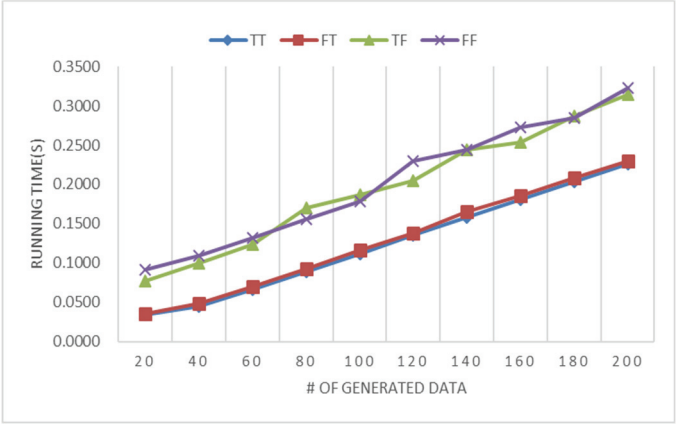

Fig. 2. Performance results of caching schemes.

7. Conclusion

In this study, we proposed a regular-expression-based DGL for synthetic big data generation. Because standard regex lacks the support of data domains, it is only suitable for defining simple patterns in a text. In this article, we introduced important enhancements to a standard regular regression to make it a powerful DGL. By introducing the type/format inference, domain, and resource reference, various types of meaningful pseudo-data can be generated.

We demonstrated the strength of the proposed language by specifying and populating the database of the TPC-H benchmark. We showed that our DGL can specify data values for all single columns as well as the relationships between tables that satisfy foreign key constraints. Moreover, it can generate inter-table correlations with skew, which is essential in the performance evaluation of join operations. In addition, the performance improvement using a cache was considered and verified through preliminary experiments.

Future work will include the automatic generation of regular expressions from examples, where regular expressions for synthetic data can be learned from positive instances. Another direction will be to implement a probability distribution estimation from sensitive datasets and synthesize mock datasets that preserve important properties in the original datasets [34-36]. This is important for privacy-preserved data mining, where real data cannot be directly obtained [37-39].

Acknowledgement

This study was supported by a grant from the JSPS Grants-in-Aid for Scientific Research (No. 20K11836) to Kai Cheng.

References

[1] A. Adir, R. Levy, and T. Salman, "Dynamic test data generation for data intensive applications," in *Hardware and Software: Verification and Testing*. Heidelberg, Germany: Springer, 2012, pp. 219-233.

[2] T. S. Buda, T. Cerqueus, J. Murphy, and M. Kristiansen, "VFDS: an application to generate fast sample databases," in *Proceedings of the 23rd ACM International Conference on Information and Knowledge Management*, Shanghai, China, 2014, pp. 2048-2050.

[3] T. Rabl, M. Frank, H. M. Sergieh, and H. Kosch, "A data generator for cloud-scale benchmarking," in *Performance Evaluation, Measurement and Characterization of Complex Systems*. Heidelberg, Germany: Springer, 2011, pp. 41-56.

[4] T. Rabl, M. Danisch, M. Frank, S. Schindler, and H. A. Jacobsen, "Just can't get enough: synthesizing big data," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, Melbourne, Australia, 2015, pp. 1457-1462.

[5] K. Taneja, Y. Zhang, and T. Xie, "MODA: automated test generation for database applications via mock objects," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, Antwerp, Belgium, 2010, pp. 289-292.

[6] H. Wu, Y. Ning, P. Chakraborty, J. Vreeken, N. Tatti, and N. Ramakrishnan, "Generating realistic synthetic population datasets," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 12, no. 4, pp. 1-22, 2018.

[7] K. Mason, S. Vejdan, and S. Grijalva, "An ″on the fly″ framework for efficiently generating synthetic big data sets," in *Proceedings of 2019 IEEE International Conference on Big Data (Big Data)*, Los Angeles, CA, 2019, pp. 3379-3387.

[8] B. C. Fung, K. Wang, R. Chen, and P. S. Yu, "Privacy-preserving data publishing: a survey of recent developments," *ACM Computing Surveys*, vol. 42, no. 4, pp. 1-53, 2010.

[9] M. Elliot and J. Domingo-Ferrer, "The future of statistical disclosure control," 2018 [Online]. Available: https://arxiv.org/abs/1812.09204.

[10] A. Dries, "Declarative data generation with problog," in *Proceedings of the 6th International Symposium on Information and Communication Technology (SoICT)*, Hue City, Vietnam, 2015, pp. 17-24.

[11] D. C. Ince, "The automatic generation of test data," *The Computer Journal*, vol. 30, no. 1, pp. 63-69, 1987.

[12] J. E. Hoag and C. W. Thompson, "A parallel general-purpose synthetic data generator," *ACM SIGMOD Record*, vol. 36, no. 1, pp. 19-24, 2007.

[13] L. Burnett, K. Barlow-Stewart, A. L. Proos, and H. Aizenberg, "The "GeneTrustee": a universal identification system that ensures privacy and confidentiality for human genetic databases," *Journal of Law and Medicine*, vol. 10, no. 4, pp. 506-513, 2003.

[14] H. Surendra and H. S. Mohan, "A review of synthetic data generation methods for privacy preserving data publishing," *International Journal of Scientific & Technology Research*, vol. 6, no. 3, pp. 95-101, 2017.

[15] A. Dandekar, R. A. M. Zen, and S. Bressan, "Comparative evaluation of synthetic data generation methods," 2017 [Online]. Available: https://sgcsc.sg/wp-content/uploads/sites/10/2020/05/RF-04.pdf.

[16] J. Fan, T. Liu, G. Li, J. Chen, Y. Shen, and X. Du, "Relational data synthesis using generative adversarial networks: a design space exploration," *Proceedings of the VLDB Endow*ment, vol. 13, no. 11, pp. 1962-1975, 2020.

[17] R. Malhotra and M. Garg, "An adequacy based test data generation technique using genetic algorithms," *Journal of Information Processing Systems*, vol. 7, no. 2, pp. 363-384, 2011.

[18] S. Sabharwal and M. Aggarwal, "Test set generation for pairwise testing using genetic algorithms," *Journal of Information Processing Systems*, vol. 13, no. 5, pp. 1089-1102, 2017.

[19] P. Fisher, N. Aljohani, and J. Baek, "Generation of finite inductive, pseudo random, binary sequences," *Journal of Information Processing Systems*, vol. 13, no. 6, pp. 1554-1574, 2017.

[20] J. Kwak and Y. Sung, "Path generation method of UAV autopilots using max-min algorithm," *Journal of Information Processing Systems*, vol. 14, no. 6, pp. 1457-1463, 2018.

[21] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, MN, 1994, pp. 243-252.

[22] J. M. Stephens and M. Poess, "MUDD: a multi-dimensional data generator," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 1, pp. 104-109, 2004.

[23] N. Bruno and S. Chaudhuri, "Flexible database generators," in *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, Trondheim, Norway, 2005, pp. 1097-1107.

[24] R. Cox, "Regular expression matching can be simple and fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)," 2007 [Online]. Available: https://swtch.com/~rsc/regexp/regexp1.html.

[25] M. D. McIlroy, "Enumerating the strings of regular languages," *Journal of Functional Programming*, vol. 14, no. 5, pp. 503-518, 2004.

[26] K. Thompson, "Programming techniques: regular expression search algorithm," *Communications of the ACM*, vol. 11, no. 6, pp. 419-422, 1968.

[27] M. Poess and C. Floyd, "New TPC benchmarks for decision support and web commerce," *ACM SIGMOD Record*, vol. 29, no. 4, pp. 64-71, 2000.

[28] A. Crolotte and A. Ghazal, "Introducing skew into the TPC-H benchmark," in *Topics in Performance Evaluation, Measurement and Characterization*. Heidelberg, Germany: Springer, 2012, pp. 137-145.

[29] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," in *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, Vancouver, Canada, 1992, pp. 27-40.

[30] E. Lo, N. Cheng, W. W. Lin, W. K. Hon, and B. Choi, "MyBenchmark: generating databases for query workloads," *The VLDB Journal*, vol. 23, pp. 895-913, 2014.

[31] M. O. Rabin and D. Scott, "Finite automata and their decision problems," *IBM Journal of Research and Development*, vol. 3, no. 2, pp. 114-125, 1959.

[32] M. Cognetta, Y. S. Han, and S. C. Kwon, "Incremental computation of infix probabilities for probabilistic finite automata," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium, 2018, pp. 2732-2741.

[33] Github, "ReverseRegex: use regular expressions to generate text strings," 2020 [Online]. Available: https://github.com/icomefromthenet/ReverseRegex.

[34] H. Ping, J. Stoyanovich, and B. Howe, "Datasynthesizer: privacy-preserving synthetic datasets," in *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, Chicago, IL, 2017, pp. 1-5.

[35] J. Drechsler, "Using support vector machines for generating synthetic datasets," in *Privacy in Statistical Databases*. Heidelberg, Germany: Springer, 2010, pp. 148-161.

[36] G. Caiola and J. P. Reiter, "Random forests for generating partially synthetic, categorical data," *Transactions on Data Privacy*, vol. 3, no. 1, pp. 27-42, 2010.

[37] X. Wu, Y. Wang, S. Guo, and Y. Zheng, "Privacy preserving database generation for database application testing," *Fundamenta Informaticae*, vol. 78, no. 4, pp. 595-612, 2007.

[38] J. Zhang, G. Cormode, C. M. Procopiuc, D. Srivastava, and X. Xiao, "Privbayes: private data release via Bayesian networks," in *Proceedings of International Conference on Management of Data (SIGMOD)*, Snowbird, UT, 2014, pp. 1423-1434.

[39] N. C. Abay, Y. Zhou, M. Kantarcioglu, B. Thuraisingham, and L. Sweeney, "Privacy preserving synthetic data release using deep learning," in *Machine Learning and Knowledge Discovery in Databases*. Cham, Switzerland: Springer, 2019, pp. 510-526.

Kai Cheng **https://orcid.org/0000-0002-5807-1691**

He received his B.S. degree from the School of Computer Science at Nanjing University in 1988 and his Ph.D. from the Graduate School of Informatics in 2002 at Kyoto University. Since 2003, he has been at the Department of Information Science, Kyushu Sangyo University. He is now a full professor and dean of the Graduate School of Information Science. His current research interests include bigdata analytics, temporal databases, and graph databases.

Keisuke Abe **https://orcid.org/0000-0003-4840-420X**

He received his B.S. degree in engineering from the University of Tokyo in 1980, M.S. degree in engineering from the University of Tokyo in 1982, and Ph.D. in engineering from the University of Tokyo in 1993. He is currently a professor in the Department of Information Science, Kyushu Sango University. His research interests include operations research and data analytics.