A Regular Expression-based DGL for Meaningful Synthetic Data Generation

Kai Cheng
Department of Information Science
Kyushu Sangyo University, Fukuoka City, Japan
chengk@is.kyusan-u.ac.jp

***Abstract*— Synthetic datasets are necessary for performance evaluation and function test in most database applications. In this paper, we propose a regular expression-based data generation language (DGL) for flexible test data generation. We extend the standard regular expressions to include references to external resources, sequential numbers, probability distributions, type/format inference, and dictionary sampling. In order to implement the proposed scheme efficiently, result caching and database caching techniques are developed and evaluated by experiments.**

Keywords—synthetic data generation, data generation language (DGL), regular expression, finite automaton, performance analysis, type/format inference

I. Introduction

In recent years, tools for populating the database with meaningful data that satisfy database constraints and statistical distributions play an increasingly important role in the development of database applications[2][12][20]. Since real data is typically subject to privacy regulations, synthetic data is a feasible solution in the development phrase. There are a number of previous work on automated data generation [2][9][10][12]. These studies however are limited in the types of data that can be generated and the obtained text data are often meaningless random strings.

A regular expression is a sequence of characters that define patterns in text. A pattern consists of one or more character literals, operators, or constructs. Each character in a regular expression is either a meta-character, having a special meaning, or a regular character that has a literal meaning. Typically, these patterns are primarily used to find matchings within a large body of text. Character sets are one of the most commonly used features of regular expressions. By placing the characters to match between square brackets, a character set is defined where any character in it can be matched. A series of shorthand character sets are available. For example, \d is short for [0-9] and \w represents all word characters, including ASCII letter, digit or underscore. By placing part of a regular expression inside round brackets or parentheses, one can group that part of the regular expression together. This allows to apply a quantifier to the entire group or to restrict alternation to part of the whole regular expression.

In this paper, we propose to use regular expressions reversely as a data generation language (DGL) for test data generation. Given a regular expression, a set of strings that exactly match it will be generated. For example, /090-\d{4}-\d{4}/ defines the pattern of the typical mobile phone numbers in Japan. So 090-1234-5678, 090-2345-5432 can be generated as instances of this pattern.

The standard regular expressions however are not sufficient as a data generation language. In this work, we introduce important extensions, such as sequential number, random number, random dictionary sampling that make regular expressions to be a powerful tool for generating realistic datasets with meaningful content and probability distributions that match the target database.

II. Extended Regular Expression as DGL

A. Data Types and Formats

In order to populate database, data should be carefully formatted to data types supported by the underlying database system. Instead of explicit data types, we introduce implicit type and format inference. Data types to be considered are as follows.

- **Integer**: numbers like 23000 are treated as integers by type inference. We also support printf-like format such as "%03d" for 001.
- **Decimal**: numbers like 0.24, 1.00 are regarded as decimal with format "%.2f"
- **Date**: 2019-12-01 is inferred as date type with format "yyyy-mm-dd" and 2019-12 is also valid date with format "yyyy-mm".
- **Time:** 12:30:00 has a time type add format "hh:mm:ss", and 12:30 is similarly a time of format "hh:mm".
- **Datetime:** 2019-12-01 12:30 is a dateime of format "yyyy-mm-dd hh:mm"

String is the default data type whenever other data types cannot be correctly inferred.

B. Primitive Generators

The most important extension to regular expressions for data generation is the introduction of primitive generators.

1. Sequential number generator

A sequential number generator (or sequential generator) produce arithmetic sequence to support identifiers as in most database systems. A sequential number generator is defined as follows.

$$range(min, max, step=1) \quad (1)$$

Identify applicable funding agency here. If none, delete this text box.

It has three arguments where *min* and *max* specify the range from which a number is generated. The third is the step to get next number, 1 by default. Based on *min* and *max*, type and format inference will decide a type and/or format for the numbers to be generated. For example,

range(1,255) outputs 1,2,3,4,…,255

range(001,255) outputs 001, 002, 003,…, 255

range('2019-1-1','2019-8-31', 7) outputs

2019-1-1, 2019-1-8, 2019-1-15, …

This is useful in the case when format is important. We will give more examples for type/format inference.

2. Random number generator

A random number generator (or random generator) is similar to sequential generator except that numbers are generated uniformly at random or follow some statistical distribution. A random number generator can be any of the following forms.

random(min, max, dist=0) (2)

There are three arguments where *min* and *max* define the range from which a number is generated. The third argument specifies a distribution by ID:

0: uniform distribution (default);

1: normal distribution;

2: exponential distribution;

3: Poisson distribution;

4: Zipfian distribution.

Based on *min* and *max,* type and format inference will decide a type and/or format for the numbers to be generated. For example,

random(1, 255) outputs 57, 2, 124, ...

random(001, 255) outputs 057, 002, 124, ...

random('9:00', '12:10') outputs 10:24, 11:30, 9:40, …

Another form of random generator is a step function defined as.

```
random(                                   (3)
  [min_1, max_1] : prob_1,
  [min_2, max_2] : prob_2,
    :
  [min_n, max_n] : prob_n,
)
```

With probability of *prob_i*, a number is drawn from *[min_i, max_i]*, here *prob_1+prob_2+…prob_n = 1*. For example,

```
random(
   [0, 59]: 0.3,
   [60, 69]: 0.2,
   [70, 79]: 0.25,
   [80,100]: 0.25
) outputs 74, 23, 89, 66, 95, ...
```

3. Random sample generator

A random sample generator (or sample generator) is a more general and powerful generator which takes any sets as input, instead of ranges as in random generator of formula (3).

```
random(                                   (4)
  set_1: prob_1,
  set_2: prob_2,
    :
  set_n: prob_n
)
```

Here with probability *prob_i*, a number is drawn from *set_i*, (i=1, 2, 3, …), here *prob_1+prob_2+…prob_n=1*. For example,

```
random(
   [S]:  0.10,
   [AB]: 0.35,
   [C]:  0.40,
   [DE]: 0.15
)
outputs A, C, B, S, D, B, ...
```

C. Dictionaries

In standard regular expressions, only ASCII characters can be used so text strings are limited to meaningless random strings. We introduce dictionaries as a new source of vocabulary. A dictionary is simply a named list of strings. For example,

Color: red, green, blue, yellow, pink, gray, white, black,…

FamilyName: Stewart, Morgan, Trump, Bush, Scott, …

GirlsName: Lucy, Lily, Sophia, Isabella, Oliva, Alice, …

BoysName: James, Oliver, Benjamin, Jackson, Henry, …

StateName: Arizona, California, Florida, Illinois, Iowa, …

StateAbbr: AZ, CA, FL, IL,IA, KY, MD, …

Dictionaries are objects maintained externally in databases or files and can be referenced by **Dict** wrapper, for example, *Dict.Color*, *Dict.FamilyName*. Dictionaries can be imported by wrapping the values in an existing database, which is useful in generating foreign keys in a table.

In regular expressions, dictionaries are used with the reference mechanism to be described in the next section.

D. References

One limit of regular expressions is that its vocabulary is restricted. It is desirable to include dictionaries from outside. To this end, we introduce references to predefined dictionaries. The following symbols in a regular expression are used as references to predefined dictionaries.

%1 ~ %9, %a ~ %z

The following regular expression defines a list of names with birthdays.

/(%1|%2) %3 , %a/

%1 := Dict.BoysName

%2 := Dict.GirlsName

%3 := Dict.FamilyName

%a := *random(*

'1980-1-1',

'1999-12-31'

)

will output:

Douglas Mitchell, 1982-4-3

Jennifer Stewart, 1990-3-14

Ernest Morgan, 1984-7-4

Isla Scott, 1999-1-15

Jessica Simmons, 1988-6-8

Sophia Moore, 1998-4-17

Susan Smith, 1997-5-1

In this example, last names are given by (%1|%2), resulting in a union of boy's names and girl's names. Using sample generator, we can decide the probabilities for boy's names and girl's names to be included. This is also known as prob union as in [2]. For example,

/%a %b/

%a := Dict.FamilyName

%b := *random(*

%1: 0.35,

%2: 0.65

),

%1 := Dict.GirlsName

%2 := Dict.BoysName

will take more boy's names than girl's names to generate names. Note that (%1|%2) is equivalent to random(%1: 0.5, %2: 0.5).

Reference is a powerful tool for populating database with reference constraints where foreign keys are generated with reference to keys in existing tables. We will describe this in details in the section.

III. Case Study: Generating TPC-H Benchmark

To demonstrate the capability of the proposed DGL, we describe how to use it to populate the database of TPC-H benchmark[17]. The TPC-H benchmark proved to be successful in the decision support area. Many commercial database vendors and their related hardware vendors used these benchmarks to show the superiority and competitive edge of their products. TPC-H consists of separate and individual tables (the Base Tables) and relationships between columns in these tables. The data types in TPC-H include Identifier, Integer, [Big] Decimal, Fixed text, Variable text, Date, all can be easily translated into DGL data types.

1. Using Dictionaries in TPC-H

TPC-H uses a few dictionaries for generating meaningful text strings. For example,

P_TYPE is a combination of words from three dictionaries as follows: PartSize={Standard, Small, Medium, Large, Economy, Promo}, PartCoat={Anodized, Burnished, Plated, Polished, Brushed} and PartMaterial={Tin, Nickel, Brass, Steel, Copper}. The following regular expression defines values for P_TYPE:

/%1\s%2\s%3/

%1:=Dict.PartSize

%2:=Dict.PartCoat

%2:=Dict.ParMaterial

The output will be strings like 'Small Plated Brass', 'Economy Burnished Copper'.

P_CONTAINER is generated by the concatenation of 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:

/%1\s%2/

%1:=Dict.ContainerSize

%2:=Dict.ContainerType

The output will be strings like 'Med Can', 'Wrap Jar'.

2. Populating Tables in TPC-H

It is very easy to define single values for individual columns in TPC-H tables. By reference and dictionary mechanisms in the proposed DGL, it is also possible to generate inter-table dependencies between columns with respect to foreign keys. In this following, we show the table definition with DGL expressions for each column and relationship between tables.

a. PART Table (SF: Scaling Factor)

· **P_PARTKEY**: identifier, ***range(1, SF*200000)***

- **P_NAME**: variable text, size 55, **/\w{10,55}/**
- **P_MFGR**: fixed text, size 25, **/\w{25}/**
- **P_BRAND**: fixed text, size 10, **/\w{10}/**
- **P_TYPE**: variable text, size 25, **/%1\s\2\s%3/,**

  %1:=Dict.PartSize, %2:=Dict.PartCoat, %3:=Dict. PartMaterial
- **P_SIZE:** integer, ***random(1,100)***
- **P_CONTAINER:** variable text, size 10, **/%1\s\2/**

  %1:=Dict.ContainerSize, %2:=Dict.ContaienrType
- **P_RETAILPRICE**: decimal, ***random(0.00, 10000.00)***
- P_COMMENT: variable text, size 23 , **/\w{1,23}/**
- Primary Key: P_PARTKEY

b. SUPPLIER Table

- **S_SUPPKEY**: identifier , ***range(1, SF*10000)***
- **S_NAME**: fixed text, size 25, **/\w{25}/**
- **S_ADDRESS**: variable text, size 40, **/\w{10,40}/**
- **S_NATIONKEY**: Foreign Key to N_NATIONKEY, **Dict.NationKey**
- **S_PHONE**: fixed text, size 15, **/%1-%2-%3-%4/**

  %1:=*random(1,999)*, %2,%3:=*random(100,999)*, %4 :=*random(1000,9999)*,
- **S_ACCTBAL**: decimal, ***random(100.00, 100000.00)***
- **S_COMMENT**: variable text, size 101, **/\w{1,101}/**
- Primary Key: S_SUPPKEY

c. PARTSUPP Table

- **PS_PARTKEY**: Identifier Foreign Key to P_PARTKEY

  Dict.P_PARTKEY
- P**S_SUPPKEY**: Identifier Foreign Key to S_SUPPKEY

  Dict.S_SUPPKEY
- **PS_AVAILQTY**: integer, ***random(1,10000)***
- **PS_SUPPLYCOST**: Decimal, ***random(0.00, 100000.00)***
- **PS_COMMENT**: variable text, size 199 , **/\w{1,101}/**
- Primary Key: **PS_PARTKEY, PS_SUPPKEY**

random(

%1: 35%

%2: 65%

),

%1:=Dict.P_PARTKEY, %2:=Dict.S_SUPPKEY

The above definition describes the probabilistic correlation between foreign keys with 35% from PART and 65% from SUPPLIER. The skewness is an important factor in evaluating join performance[7]. As pointed in [5], while uniform data distributions were a design choice for the TPC-D benchmark and its successor TPC-H, it has been universally recognized that data skew is prevalent in data warehousing. A modern benchmark should therefore provide a test bed to evaluate the ability of database engines to handle skew. The proposed DGL is suitable for generating data with skews.

IV. Implementation

To implement the proposed DGL for data generation, a system should consist of the following components (Figure 1).

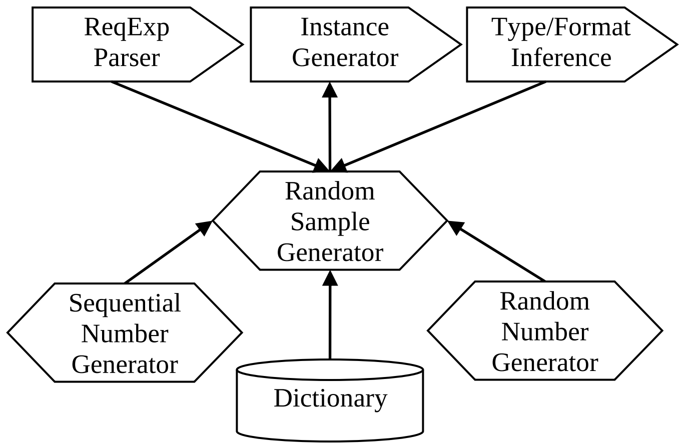

Figure 1. System architecture for DGL implementation

RegExp parser takes a regular expression as input, analyzes its syntax structure and builds a sequence of elements in the form <character_set, repetition>.

Type/format inference determines the data type, such as int, float, date, time, and format of instances to be generated by analyzing the literals given in the arguments. For example, 000 implies an integer type in the form of %03d (as defined in printf of C). That is, integer of fixed size 3 padded with leading 0s if necessary.

Sequential number generator is used for generating arithmetic sequences where the difference between one term and the next is a constant. It is useful for identifiers systematically.

Random number generator is used for generating random numbers, which can be uniformly distributed over the range, or following some statistical distribution, such as normal distribution, exponential distribution.

Random sample generator plays a crucial role in combining instances from multiple sources. In the simplest case, it produces samples from a set or a dictionary. When specifying

The instance generator plays a central role in generating synthetic data. There are two methods for regular expression matching: non-deterministic automaton (NFA) and deterministic automaton (DFA). In regular expression matching by sequential calculation, it is often more efficient to use DFA to determine whether or not to match. However, the number of states of DFA may be exponential times the number of states of NFA corresponding to the same regular expression, and in that case, it is more efficient to use NFA.

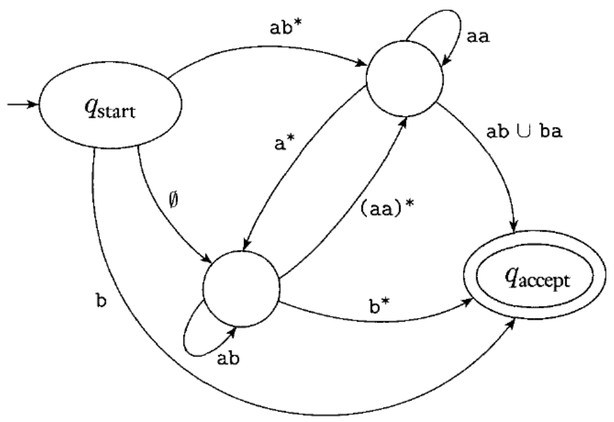

Figure 2 Regular expresion and gemeralized NFA

In NFA and DFA, one transition is made for each input symbol. A computer model that relaxes this restriction is the extended nondeterministic finite automaton (GNFA). In GNFA, each state transition is an NFA corresponding to an arbitrary regular expression. GNFA reads multiple characters at once from the input, but the string corresponds to the regular expression attached to the transition (edge). GNFA can be easily converted to regular expressions. The conversion converts intermediate transitions into regular expressions, and finally makes a single transition from the starting state to the accepting state (Figure 2). Similarly, the regular expression added to each GNFA transition can be converted to NFA by adding intermediate states until it is decomposed into single characters.

Table 1 Regular expresion and gemeralized NFA

	RegExp Caching	DB Caching
TT	Yes	Yes
FT	No	Yes
TF	Yes	No
FF	No	No

V. Performance Evaluation

To improve the efficiency of data generation, it is effective to use caching technology. (1) Sub-automata and partial regular expression caching, and (2) DB caching can be expected to improve data generation efficiency.

(1) Sub-automata and partial regular expression 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, saving 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 the DBMS standard caching technology can be used, it is necessary to shuffle the cache data so that the same result is not obtained each time the cache is used.

The evaluation experiments were conducted to evaluate the efficiency of the two caches. The evaluation targets are shown in Table 1. The change in execution time was examined depending on whether the regular expression analysis result cache (RegExp Caching) and the database cache (DB Caching) were used.

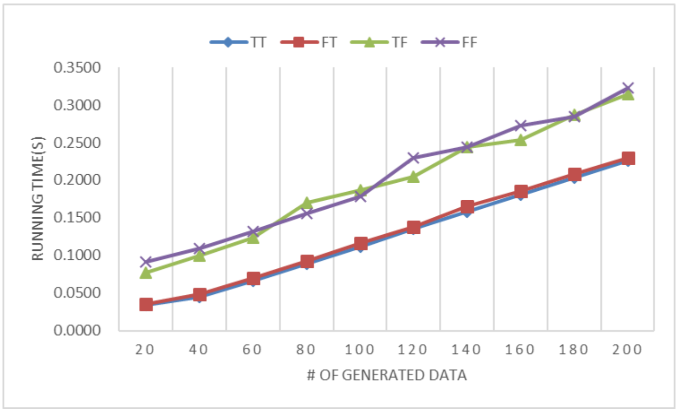

Figure 3 Performance results of caching schemes

Figure 3 shows the experimental results. Regular expression analysis uses the PHP package ReverseRegex[24]. The execution time is the average of 10 repetitions. It was shown that the use of a DB caching is more efficient than RegExp caching in reducing execution time.

VI. Conluding Remarks

In this paper, we have proposed a regular expression-base DGL for database generation that can generate random data according to probability distribution and realistic pseudo data. Values with some patterns, such phone numbers, zip codes for individual columns can be defined using regular expressions. By introducing type/format inference, dictionary and the extended reference mechanism , various kinds of meaningful pseudo data can be generated.

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

The future work includes automatic generation of regular expressions from example data, where regular expressions for synthetic data can be learned from positive instances. This is important for privacy-preserved data mining where real data cannot be obtained directly.

REFERENCES

[1] Adir , R. Levy , T. Salman, Dynamic test data generation for data intensive applications, Proceedings of the 7th international Haifa Verification conference on Hardware and Software: verification and testing, December 06-08, 2011, Haifa, Israel

[2] N. Bruno and S. Chaudhuri. Flexible database generators. In Proceedings of the 31st international conference on Very large data bases (VLDB), pages 1097-1107, 2005.

[3] T. S. Buda, T. Cerqueus , J. Murphy , M. Kristiansen, VFDS: An Application to Generate Fast Sample Databases, Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, November 03-07, 2014, Shanghai, China

[4] R. Cox Regular Expression Matching can be simple and fast (but is slow in Java, Perl, PHP, Python, Ruby, ...), (January 2007) https://swtch.com/~rsc/regexp/regexp1.html

[5] A. Crolotte, A. Ghazal (2012) Introducing Skew into the TPC-H Benchmark. In: Nambiar R., Poess M. (eds) Topics in Performance Evaluation, Measurement and Characterization. TPCTC 2011. Lecture Notes in Computer Science, vol 7144. Springer, Berlin, Heidelberg

[6] R.A. DeMillo, A.J. Offutt. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering. 19: 640. September 1991

[7] D.J. DeWitt, J. F. Naughton, D. A. Schneider, S. Seshadri.: Practical Skew Handling in Parallel Joins. In: Proceedings of VLDB 1992, pp. 27–40 (1992)

[8] M. Douglas McIlroy, Enumerating the strings of regular languages, Journal of Functional Programming 14 (2004), pp. 503–518

[9] A. Dries, Declarative Data Generation with ProbLog, in Proc. of the Sixth International Symposium on Information and Communication Technology (SoICT 2015), pp/17-24, 2015

[10] D. C. Ince, The automatic generation of test data, Comput. J., 30, 63-69 (1987).

[11] Lo, E., Cheng, N., Lin, W. W., Hon, W. K., & Choi, B. MyBenchmark: generating databases for query workloads. The VLDB Journal—The International Journal on Very Large Data Bases, 23(6), 895-913,2014

[12] Hoag, Joseph E., and Craig W. Thompson. A parallel general-purpose synthetic data generator. Data Engineering. Springer, Boston, MA, 2009. 103-117.

[13] Korel. A Dynamic approach of automated test data generation. Conference on Software Maintenance. (1990)

[14] K. Pan , X. Wu , T. Xie, Generating program inputs for database application testing, Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, p.73-82, November 06-10, 2011

[15] K. Pan, X. Wu, T. Xie, Automatic test generation for mutation testing on database applications, Proceedings of the 8th International Workshop on Automation of Software Test, May 18-19, 2013, San Francisco, California

[16] R. Pargas,M. Harrold, R. Peck. Test Data Generation using Genetic Algorithms. Journal of Software Testing, Verification and Reliability. 9: 263–282 (1999)

[17] M. Pöss, C. Floyd, New TPC Benchmarks for Decision Support and Web Commerce, SIGMOD Record, 29(4): 64-71, 2000

[18] M. Rabin and Dana Scott, Finite automata and their decision problems, IBM Journal of Research and Development 3 (1959), pp. 114–125.

[19] T. Rabl, M. Frank, H. M. Sergieh, and H. Kosch. A data generator for cloud-scale benchmarking. In Proceedings of the Second TPC technology conference on Performance evaluation, measurement and characterization of complex systems (TPCTC), pages 41-56, 2011.

[20] Rabl, Tilmann, et al. Just can't get enough: Synthesizing Big Data. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015.

[21] K. Taneja, Y. Zhang, and T. Xie. Moda: Automated test generation for database applications via mock objects. In Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE 2010), short paper, 2010.

[22] K. Thompson, Regular expression search algorithm, Communications of the ACM 11(6) (June 1968), pp. 419–422. http://doi.acm.org/10.1145/363347.363387

[23] X. Wu, Y. Wang, S. Guo, and Y. Zheng. Privacy preserving database generation for database application testing. Fundam. Inf., 78(4):595-612, Dec. 2007.

[24] ReverseRegex : Use Regular Expressions to generate text strings https://github.com/icomefromthenet/ReverseRegex