ReX: Extrapolating Relational Data in a Representative Way

Teodora Sandra Buda, Thomas Cerqueus, John Murphy, Morten Kristiansen

ReX: Extrapolating Relational Data in a Representative Way

Teodora Sandra Buda[1], Thomas Cerqueus[2], John Murphy[1], and Morten Kristiansen[3]

[1] Lero, Performance Engineering Lab
School of Computer Science and Informatics, University College Dublin.
teodora.buda@ucdconnect.ie, j.murphy@ucd.ie
[2] Université de Lyon, CNRS, INSA-Lyon, LIRIS, UMR5205, F-69621, France.
thomas.cerqueus@insa-lyon.fr
[3] IBM Collaboration Solutions, IBM Software Group, Dublin, Ireland.
morten_kristiansen@ie.ibm.com

Abstract. Generating synthetic data is useful in multiple application areas (e.g., database testing, software testing). Nevertheless, existing synthetic data generators generally lack the necessary mechanism to produce realistic data, unless a complex set of inputs are given from the user, such as the characteristics of the desired data. An automated and efficient technique is needed for generating realistic data. In this paper, we propose ReX, a novel extrapolation system targeting relational databases that aims to produce a representative extrapolated database given an original one and a natural scaling rate. Furthermore, we evaluate our system in comparison with an existing realistic scaling method, UpSizeR, by measuring the representativeness of the extrapolated database to the original one, the accuracy for approximate query answering, the database size, and their performance. Results show that our solution significantly outperforms the compared method for all considered dimensions.

Keywords: Representative extrapolation, Scaling problem, Synthetic data generation, Relational database.

1 Introduction

Generating synthetic data is convenient in multiple application areas (e.g., software validation, data masking, database testing). Synthetic data is generally used when real data is not available, when it cannot be published publicly or when larger amounts of data are needed. Therefore, it represents an artificial enabler for any analysis that requires data. When using synthetic data, a necessary evaluation is how representative it is in comparison to real-life data.

Extrapolating the data from an existing relational database is a potential solution to overcome the lack of realism of the synthetic data. There are two directions that can be explored for scaling data: (i) to a particular size, or (ii) to a particular time in the future. The first is useful in multiple application areas

where the size of the generated database matters, such as scalability testing. The second direction could be addressed by applying machine learning techniques to predict how data will evolve using accurate historical data. In this paper, we explore the first path, which represents a starting point for studying the evolution of a database. Maintaining the distributions present in the original database contributes to the realism of the generated data. The representativeness dimension is crucial as the results of the analysis to be applied on the representative extrapolated database are expected to be similar to the ones from the original database (e.g., in approximate query answering). This path has been explored before. In [19], the authors introduce the scaling problem as follows:

Scaling Problem *Given a relational database D and a scaling factor s, generate a database D' that is similar to D but s times its size.*

The authors propose a novel tool, namely UpSizeR, which aims to solve the scaling problem in an innovative way, using mining algorithms such as clustering to ensure that the representativeness is maintained. The method requires complex inputs from the user (e.g., the probability perturbation exponent). Most of the existing synthetic database generators require complex inputs from the user in order to generate realistic data [3, 11, 1]. However, complex inputs require expert knowledge, and thus may lead to poor accuracy in the results.

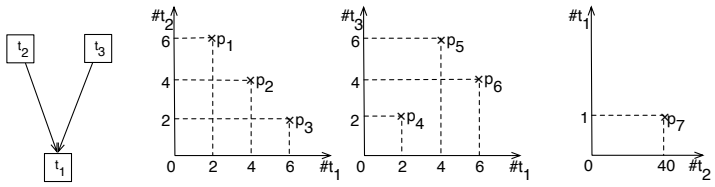
In this paper, we propose an automated representative extrapolation technique, ReX, that addresses the scaling problem above. Similarly to [4] and [19], we define a representative database as a database where the distributions of the relationships between the tables are preserved from the original database. As foreign keys are enforced links between tables, they represent invaluable inputs to depict the relationships between data in a relational database. This represents a first step towards achieving a representative extrapolated database. We devise two techniques for handling non-key attributes. To illustrate ReX's applicability in a real scenario, we perform approximate query answering evaluation. We compare ReX to UpSizeR [19] and show that our solution outperforms UpSizeR in terms of representativeness, query answering, database size, and execution time.

The remainder of this paper is organized as follows: Section 2 introduces the potential solutions to the scaling problem. Section 3 presents the representative extrapolation system, ReX. Sections 4 presents the evaluation of ReX. Section 5 presents the related work. Finally, Section 6 concludes the paper.

2 Potential Scaling Strategies

In this section we investigate the potential directions in which relational data should grow such that it is representative of the original database.

Notations. We denote by FK_i^j the set of attributes of table t_i that reference table t_j. We denote this relationship by $t_i \rightarrow t_j$ and say that t_i and t_j are associated tables. This notation is used for constructing the graph structure of a database where an edge represents a relationship and a node represents a



(a) Schema. (b) Distributions. (c) Inverted axes.

Fig. 1: Example graph schema and distributions.

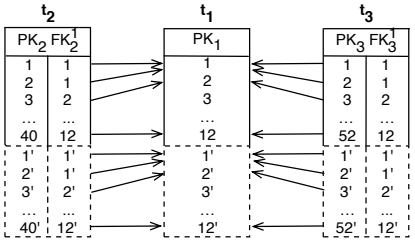


Fig. 2: Extrapolation solution, with $s = 2$.

Pos k : $(k1, k2)$	$f_0(x)$	$f_1(x)$
1	8, "a"	1, "1" 6, "6"
2	15, "b"	2, "2" 7, "7"
3	13, "c"	3, "3" 8, "8"
4	1, "g"	4, "4" 9, "9"
5	3, "e"	5, "5" 10, "10"

Table 1: $f_i(x)$ example.

table. Moreover, we refer to parents of t as the set of tables that reference t: $parents(t) = \{t_i \in T : t_i \to t\}$. In Fig. 1(a), $parents(t_1) = \{t_2, t_3\}$. Similarly, we refer to children of table t by: $children(t) = \{t_i \in T : t \to t_i\}$. For instance, $children(t_2) = \{t_1\}$. A table with no children is called a *leaf* table (e.g., t_1). In order to determine the growth direction of a database O, we represent the relationships between each pair of tables, $\forall t_i, t_j \in T$, $t_j \to t_i$, through a scatter plot denoted by $sp_{t_i}^{t_j}$, where t_i appears on the x-axis and t_j on the y-axis. Let us consider the case study presented in Fig. 1. Figure 1(a) presents the graph-structured schema of the database O. Figure 1(b) portrays the generated scatter plots $sp_{t_1}^{t_2}$ between t_1 and t_2, and $sp_{t_1}^{t_3}$ between t_1 and t_3. A point at a coordinate (x, y) of a scatter plot $sp_{t_i}^{t_j}$ expresses that x tuples of t_i are individually referenced by y distinct tuples of t_j, and that $y \cdot x$ tuples of t_j reference x tuples of t_i. For instance, point $p_1(2, 6)$ in $sp_{t_1}^{t_2}$ indicates that two tuples of table t_1 are each individually referenced by six tuples of table t_2 (i.e., $6 \cdot 2 = 12$ tuples of t_2 reference 2 tuples of t_1). When the axes are inverted, $sp_{t_j}^{t_i}$, since $t_j \to t_i$, a point $p(x, y)$ of $sp_{t_j}^{t_i}$ expresses the x tuples of t_j reference y distinct tuples of t_i. In this case, $sp_{t_j}^{t_i}$ consists of a single point, $p(\|t_j\|, 1)$, as each tuple of t_j has a single foreign key value referencing t_i. For instance in Fig. 1(c), the scatter plot $sp_{t_2}^{t_1}$ indicates that $\|t_2\|$ tuples of t_2 are referencing a single tuple of t_1, as each tuple of t_2 contains a single reference to t_1. Through a scatter plot $sp_{t_i}^{t_j}$ we can

compute the number of tuples of t_i and t_j from O, $\|O(t_i)\|$ and $\|O(t_j)\|$, with:

$$\|O(t_i)\| = \sum_{\forall p(x,y) \in sp_{t_i}^{t_j}} x, \text{ and } \|O(t_j)\| = \sum_{\forall p(x,y) \in sp_{t_i}^{t_j}} (y \cdot x)$$

From Figure 1(b), we determine that: $\|O(t_1)\| = 12$, $\|O(t_2)\| = 40$, and $\|O(t_3)\| = 52$. When extrapolating O by s to produce the extrapolated database X, we expect that each table t of O will be scaled in size by s such that: $\|X(t)\| = s{\cdot}\|O(t)\|$.

A **horizontal** growth direction for each point of a scatter plot produces the optimal results in terms of database size. Considering a horizontal growth direction, each point p of $sp_{t_i}^{t_j}$ scales s times on the x-axis: $\forall p(x, y)$ becomes $p'(x', y)$, where $x' = s \cdot x$. This leads to the following properties of t_i and t_j in X:

$$\|X(t_i)\| = \sum_{\forall p(x',y) \in sp_{t_i}^{t_j}} (s{\cdot}x) = s{\cdot}\|O(t_i)\|, \; \|X(t_j)\| = \sum_{\forall p(x',y) \in sp_{t_i}^{t_j}} (y{\cdot}(s{\cdot}x)) = s{\cdot}\|O(t_j)\|$$

Through horizontal scaling: $\|X(t_1)\| = 24$, $\|X(t_2)\| = 80$, and $\|X(t_3)\| = 104$. These are the desired expected sizes of the tables. This leads to X being representative of O (i.e., as each point is scaled by s), and of accurate size (i.e., as each table is scaled by s). Therefore, the extrapolation solution must create for each of the x identifiers of t_i, pk_i, exactly s-1 new identifiers, pk'_i, and for each of the $x \cdot y$ key values of t_j, (pk_j, fk_j), exactly s-1 new key values of t_j, (pk'_j, fk'_j), each individually referencing one of the s-1 new identifiers created for t_i, $fk'_j = pk'_i$. This is exemplified in Fig. 2, where $t_i = t_1$, and $parents(t_i) = \{t_2, t_3\}$.

3 ReX: Extrapolation System

In this paper, we propose a system called ReX[1] that aims to produce a representative extrapolated database X, given a scaling rate, $s \in \mathbb{N}$, and a relational database O. The objective is to maintain the distributions between the consecutive linked tables and the referential integrity of the data. We assume that there are no cycles of dependencies and that foreign keys only reference primary keys.

ReX produces the extrapolated database in a single pass over the entire original database and thus reduces the complexity of a two-step algorithm that would compute the expected scaled distribution and generate data accordingly.

Natural scale discussion. When the scaling rate is a real number (i.e., $s \notin \mathbb{N}$), the floating part requires the generation of tuples for only a fraction of each table from O. Thus, the method must decide for which partial number of tuples of t_j it should create new tuples. As this represents a different problem by itself [4, 8], in this paper we consider only natural scaling rates. Moreover, the scenario of naturally scaling databases is commonly applicable to enterprises where it is rarely needed to extrapolate to a fraction rather than a natural number. The maximum error brought by approximating the real scaling rate to a natural

[1] Representative eXtrapolation System, https://github.com/tbuda/ReX

number is 33.33%, and occurs for $s = 1.5$ (i.e., caused by X containing 33.33% less or more tuples than desired). The impact of the floating part decreases as s increases (e.g., when $s = 10.5$ the error caused by approximating is is reduced to 4.8%). Another solution is using a sampling method for the remaining fractional number. However, both solutions would introduce errors in the results, and in this paper we are interested in evaluating the extrapolation technique.

3.1 Key attributes generation

The keys generation function targets both primary and foreign keys of a table. We denote the function by $f_i : \mathbb{D}_k \to \mathbb{D}_k$, where $\mathbb{D}_k$ is the domain of the key k, and i is the iteration number, $i \in [0, s)$. The function is required to satisfy the following properties: (i) injectivity: $\forall i, j \in \mathbb{N}, \forall x_1, x_2 \in \mathbb{D}_k, x_1 \neq x_2 \Rightarrow f_i(x_1) \neq f_j(x_2)$, (ii) uniqueness between iterations: $\forall\ i, j \in \mathbb{N},\ i \neq j, \forall x \in \mathbb{D}_k$, $f_i(x) \neq f_j(x)$. ReX uses a positive arithmetic progression with a common difference of 1 (i.e., 1,2,3,...). The function receives as input a value x and the iteration number $i \in [0, s)$, and outputs a new value converted to $\mathbb{D}_k$: $f_i(x) = cast(p(x) + i \cdot \|t\|)_{\mathbb{D}_k}$, where x is a value of the key k, primary in table t, and $p(x)$ represents the position of the tuple identified by x in $O(t)$. The *cast* function converts the natural number produced by the arithmetic progression to the domain of the key. An example of $f_i(x)$ is presented in Table 1 where $T = \{t\}$, $\|O(t)\| = 5$, $\|PK_t\| = 2$, *integer* and *varchar*, and $s = 2$. When a key is composed of multiple attributes, the function is applied on each attribute, using the first position for each value across their occurrences to ensure referential integrity. Moreover, for the same key value and position, the function generates the same output. This ensures that the referential integrity is not breached as the newly generated foreign key values will reference the new primary key values.

3.2 Non-key attributes generation

ReX can perform the following operations: (1) generate **new** values for the non-key attributes either by: (i) generating synthetic values using the generation function proposed, or (ii) using a dictionary with sample values for each type of attribute, or (2) manipulate the **existing** values for the non-key attributes either by: (i) selecting a random value from the original database, (ii) selecting a random value from the original database such that the frequency count of the non-key attribute is maintained, or (iii) maintaining their original values. In this paper, we present results of ReX implemented using (2.ii) denoted further by ReX_{rfc}, and (2.iii) denoted by ReX_{main}, as these ensure that the value range constraints are not breached and that the approximate query evaluation will not be affected by the synthetic values. The first solution, ReX_{rfc}, increases the diversity of the data produced by generating random content from O, and might cover certain scenarios that the second solution would miss. Such a scenario is for instance the sudden growth of female computer scientists. This could be vital for instance in software testing, as a random selection of non-key attributes' values could cover more test cases than the original ones. Moreover, we expect that

maintaining the frequency count of the non-key attributes ensures that queries that compute an aggregate of a non-key attribute scale according to s with no errors (e.g., the maximum age entry in a *Person* table). Furthermore, the second solution, ReX_{main}, ensures that the X preserves intra-tuple correlations (e.g., between the age and marital status attributes of a *Person* table), intra-table correlations at an attribute level (e.g., between the age of a *Person* table and its balance in an *Account* table) and frequency count of non-key values.

3.3 Approach

ReX selects the leaf tables as starting tables. The algorithm maintains the position of each primary key value when populating a table using a hash table. Thus, by starting with the leaf tables, the method avoids potentially time consuming queries for determining the position of a foreign key value in its original referenced table, and retrieves is from the hash table previously constructed. Moreover, through this bottom-up approach, X is produced through a single pass over each table of O. Phase one of the algorithm consists of generating the new key and non-key attributes' values for the leaf tables. The method retrieves the records of the leaf table from O and enforces a horizontal growth direction by generating s new tuples for each tuple of a table from O. Regarding key values, ReX will call the generation function $f_i(x)$, described in Section 3.1. Regarding non-key values, ReX_{main} maintains their values from the original tuple. ReX_{rfc} randomly selects a value from $O(t_i)$, while maintaing its frequency count. This is achieved through the SQL query on O: `SELECT` nk `FROM` t_i `ORDER BY RAND()`. In order to maintain the frequency count, ReX_{rfc} runs the query s times and iterates through the result set returned, ensuring that each value has been used s times for producing X. Phase two consists of identifying the next table to be filled. The algorithm recursively fills the parents of the already populated tables until the entire database is processed. To avoid size overhead or referential breaches due to processing a table multiple times (e.g., due to diamond patterns [8]), a table can only be populated once its children have been populated.

4 Evaluation

In this section, we compare our extrapolation system ReX to the UpSizeR approach [19]. Both methods aim to construct an extrapolated database representative of the original one, that also maintains the referential integrity of the data.

UpSizeR Overview. UpSizeR represents a representative scaling method that addresses the scaling problem. Its objective is to generate synthetic data with similar distributions of the relationships between the tables of the database (i.e., between primary and foreign key pairs) to the ones from the original database [19]. For this purpose, the approach computes the relationship degree (i.e., cardinality constraint) of each existing key of each table in the original database and generates synthetic data accordingly. In the case of a table with multiple foreign key constraints, the method uses a clustering algorithm for generating

G_1	$\mathbf{F_1}$: SELECT AVG(‘Order‘.amount) FROM ‘Order‘⋈Account⋈Disposition⋈Card WHERE Card.type=’classic’; $\mathbf{F_4}$: SELECT SUM(Trans.balance) FROM Trans⋈ ⋈Account⋈Disposition⋈Card WHERE Card.type=’junior’;
G_2	$\mathbf{F_2}$: SELECT Card.card_id FROM Card⋈Disposition WHERE Disposition.type= ’OWNER’; $\mathbf{F_3}$: SELECT Loan.loan_id FROM Loan⋈Account⋈Disposition WHERE Disposition.type=’DISPONENT’; $\mathbf{F_5}$: SELECT Client.client_id FROM Client⋈ ⋈Disposition⋈Account WHERE Account.frequency=’POPLATEK MESICNE’;
G_3	$\mathbf{F_6}$: SELECT AVG(IQ.N) FROM (SELECT district_id, COUNT(account_id) AS N FROM Account GROUP BY district_id) AS IQ; $\mathbf{H_6}$: SELECT AVG(IQ.N) FROM (SELECT l_orderkey, COUNT(l_id) AS N FROM Lineitem GROUP BY l_orderkey) AS IQ;
G_4	$\mathbf{F_7}$: SELECT AVG(avg-salary) FROM District;

Table 2: Queries used for approximate query evaluation.

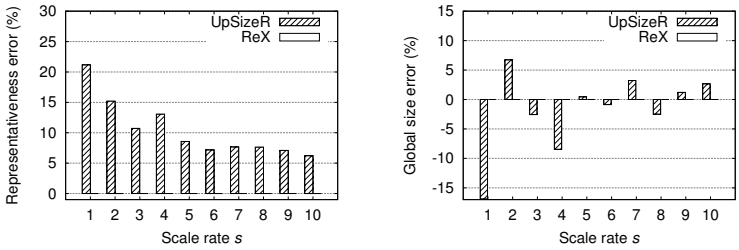
a joint degree distribution of the table. However, the mechanisms employed by UpSizeR can lead to time-consuming operations and require complex parameters as inputs from the user, which can lead to inaccurate results.

Environment and Methodology. ReX was developed using Java 1.6. ReX and UpSizeR were applied on MySQL 5.5.35 databases. They were deployed on a machine consisting of 2 Intel Xeon E5-2430 CPUs of 2.20GHz and 6 cores each, 64GB RAM, and 2TB Serial ATA Drive with 7,200rpm, running 64-bit Ubuntu 12.04. The MySQL server was run with default status variables. We used the centralized version of UpSizeR available online[2]. We assume that the user has no prior knowledge of the database to be extrapolated and keep the default parameters' values. This coincides with the evaluation strategy the authors presented in [19]. Moreover, we show in Section 4 that the default parameters provide a near optimal configuration for the database considered.

Database. We used the Financial database[3] from the PKDD'99 Challenge Discovery in order to evaluate ReX and UpSizeR in a real environment. It contains typical bank data, such as clients information, their accounts, and loans. It contains 8 tables, and a total of 1,079,680 tuples. The sizes of the tables range from 77 (*District*) to 1,056,320 tuples (*Trans*). The Financial database schema is depicted in [4]. The starting table identified by ReX is the *District* table. Moreover, we performed similar experiments using the TPC-H database, and UpSizeR showed lower errors for the criteria considered. Similar observations were drawn regarding ReX's performance compared to UpSizeR's.

Metrics. Both ReX and UpSizeR aim to scale the distributions of the relationships between tables by s (i.e., through primary and foreign keys). In [4] we proposed a sampling method that aimed to scale the same distributions by a sampling factor. We use the average **representativeness** error metric defined in [4], replacing the sampling rate with the scaling rate. Moreover, we use the **global size** error metric defined in [4] to evaluate the size of X related to O. We

[2] comp.nus.edu.sg/∼upsizer/#download
[3] lisp.vse.cz/pkdd99/Challenge/berka.htm



(a) Representativeness. (b) Global size.

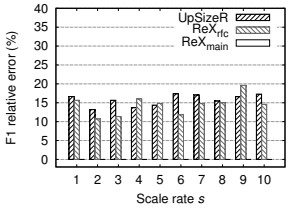
Fig. 3: Representativeness and database size errors.

measure the **query relative** error of the extrapolated database X for evaluating the query answering on X compared to O. The metric is described in detail in [5]. In this evaluation, we consider: (G_1) queries that compute an aggregate value on a non-key attribute with a `WHERE` clause on a non-key attribute (e.g., average account balance for a male client), (G_2) queries that compute an aggregate value on a key attribute with a `WHERE` clause on a non-key attribute (e.g., average number of cards for a female client), (G_3) queries that compute an aggregate value on a key attribute (e.g., average number of cards per account), and (G_4) queries that compute an aggregate value on a non-key attribute. G_3 queries investigate whether the distributions between the tables have been preserved from a query answering perspective. Moreover, G_4 queries investigate if the frequency count preservation of non-key attributes increases the accuracy of queries targeting the attributes. Table 2 presents the queries used in this evaluation. Finally, we evaluate the methods' performance by measuring their **execution time**. This represents the run time (i.e., the pre-processing phases, such as the graph construction or diamond patterns discovery, together with the extrapolation time).

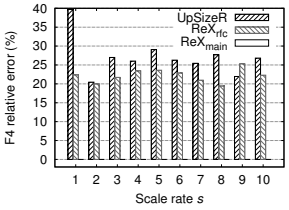
Results and observations

Representativeness. Figure 3(a) presents the results of UpSizeR and ReX (i.e., ReX_{main} and ReX_{rfc}) in terms of representativeness of the relationships between consecutively linked tables of the Financial database. We observe that UpSizeR produces an extrapolated database with the representativeness error varying between 21.2% and 6.5%, and an average of 10.5%. We observe that ReX maintains 0% error with regards to representativeness. This is because both ReX_{main} and ReX_{rfc} enforce a horizontal scaling which leads to generating for each (pk,fk) pair of each table exactly s new pairs, described in Section 2.

Database size. Figure 3(b) presents the results of UpSizeR and ReX (i.e., ReX_{main} and ReX_{rfc}) in terms of expected database size. We observe that UpSizeR's global size error varies between -16.9% and 2.7% error, with an absolute average of 4.6% on the Financial database. Moreover, we observe that ReX maintain 0% error in terms of global size errors due to horizontal scaling of each relationship, which determines scaling each table by s.



(a) F_1 query relative error.



(b) F_4 query relative error.

Fig. 4: G_1 query relative error on Financial.

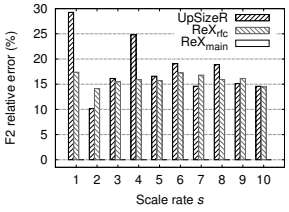


Fig. 5: F_2 query error.

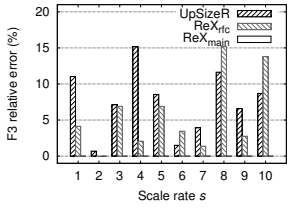


Fig. 6: F_3 query error.

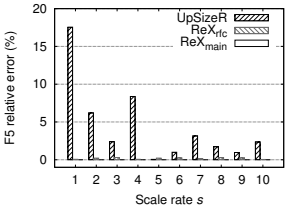


Fig. 7: F_5 query error.

Query answering. We observe in Figure 4 that UpSizeR and ReX_{rfc} show similar query answering errors on the Financial database. UpSizeR shows slightly worse results than ReX_{rfc}, with a peak error of 39.7%, occurring for F_4 when s equals 1. This is because both methods do not aim at preserving intra table correlations at a non-key attribute level, and as such, their answers are influenced firstly by their non-key attribute generation strategy and secondly by how well they preserve the representativeness of the relationships across tables. The query answering errors are expected to decrease in the case of G_2 type queries, as a single non-key attribute is involved in the `WHERE` clause of the queries. Therefore, we observe in Fig. 5 to 7 improved results of ReX_{rfc} over UpSizeR due to its precision in preserving both representativeness of the key attributes relationships and frequency count of the non-key attributes. ReX_{rfc} shows close to 0% error for F_5 query. We observe from Fig. 4 to 7 that ReX_{main} maintains 0% query relative error in terms of G_1 and G_2 queries due to horizontal scaling and maintaining the original values of the non-key attributes. Moreover, we observe a similar trend between Fig. 8 and Fig. 3(a), for the F_6 query answering and the representativeness error for UpSizeR. We notice that ReX maintains 0% error for the G_3 query answering, due to horizontal scaling. Moreover, we observe in Fig. 9 that UpSizeR shows little errors, confirming that the method considers preserving the frequency count when generating non-key attributes. We observe that ReX maintain 0% error for the G_4 query answering, due to them preserving the frequency count of the non-key attributes.

Execution time. Figure 10 presents the methods' execution time on the Financial database. We notice that ReX is up to 2 times faster than UpSizeR. When

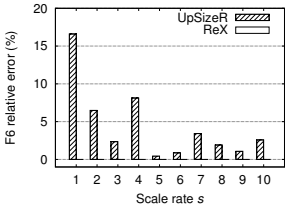


Fig. 8: G_3 query error.

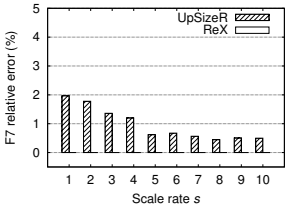


Fig. 9: G_4 query error.

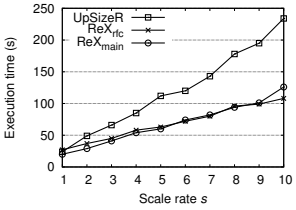


Fig. 10: Execution time.

applied on a larger database, such as a 1GB TPC-H database, we observed more significant differences between the methods' performance. In particular, ReX performed between 3 and 8.5 times faster with an average of 23 minutes difference between UpSizeR and ReX's execution run time.

Additional Discussion. When using a system with complex inputs, the challenge stands in determining the optimal parameters on the target database. We investigate the impact of the *number of clusters* expected, k (used in the generation of the joint degree distribution) and the *probability perturbation exponent*, p (used in the generation of the joint probability matrix) on UpSizeR, as they represent key inputs for UpSizeR's generation process. We considered the following set of values for k and p: {3,5,25,50,100,500,2500,5000}, and {-15,-10,-7,-5,-3,-1,0,10}, respectively. Increasing k to 5,000 raised the run time of UpSizeR to $16.4h$, compared to $12s$ when k is 3 by default. Running UpSizeR with p equal to -25 and -50 did not scale and after 10 days their execution was stopped. Identical results were found for p equal to 10, 50, and 500. The query relative error of F_7 is 1.8%, regardless of k and p. Similar conclusions were drawn for $s = \{2, 5, 8\}$ and when jointly varying k and p. Results suggest that the modification of the parameters brings little benefits for all dimensions considered. In contrast, we observe that UpSizeR's parameters have a significant impact mainly on the query answering accuracy. Small variations of the parameters resulted in high errors in query answering. This suggests that a trial and error approach might not lead to any benefits, even after a large amount of time is invested.

5 Related Work

Significant efforts have been made to improve the realism of synthetic data generators. We acknowledge them below, based on their application area.

General methods. Many commercial applications generate synthetic databases that respect the given schema constraints and use real sources as input for several attributes (e.g., names, age)[4]. Furthermore, the academic community have proposed many general-purpose synthetic data generators [12, 9, 11]. MUDD [17] is another parallel data generator that uses real data for the attributes' domain. In [3], the authors propose a *Data Generation Language* to specify and generate

[4] sqledit.com/dg, {spawner,dgmaster}.sourceforge.net, generatedata.com

databases that can respect inter and intra table correlations. However, the user must learn the specification language and input the distributions.

Software testing. Existing methods for populating testing environments usually generate synthetic data values or use some type of random distribution to select data from the production environment to be included in the resulting database [18, 14]. AGENDA [7] is a synthetic data generator based on a-priori knowledge about the original database (e.g., test case expected behavior). Furthermore, in [6] the authors describe a new approach for generating data for specific queries received as input. QAGen [2], MyBenchmark [13], and DataSynth [1] similarly generate query-aware test databases through cardinality constraints. However, they require complex inputs (e.g., distribution of an attribute, queries), which can be error-prone, as they might exclude vital test cases.

Data mining. In [15], the authors propose a synthetic data generator for data clustering and outlier analysis, based on the parameters given as input (e.g., number of clusters expected, size, shape). In [16], the authors propose a synthetic data generator that receives as input a set of maximal frequent itemset distributions and generate itemset collections that satisfy these input distributions. Other tools that can be used in this field are WEKA [10], GraphGen[5], IBM QUEST[6]. For instance, GraphGen generates synthetic graph data for frequent subgraph mining. However, the approaches require input parameters and generally produce synthetic data targeting a data mining algorithm.

6 Conclusion and Future work

In this paper, we proposed ReX, a novel automated and efficient system to representatively extrapolate a relational database, given an existing database and a natural scaling rate. The objective is to preserve the distributions of the relationships between tables and the referential integrity of the data. We presented two variations of ReX: (i) ReX_{main}, which maintains the original non-key attributes' values of the generated tuples, and (ii) ReX_{rfc} which randomly selects values for the non-key attributes from the original database such that their frequency count is preserved. We compared our technique with a representative scaling technique, UpSizeR, and showed that ReX significantly outperforms UpSizeR in representativeness and database size. Moreover, ReX is up to 2 times faster than UpSizeR. Results show that ReX is highly suitable for approximate query answering, which leads to various application scenarios, such as scalability testing. Finally, results suggest that UpSizeR is sensitive to the variation of the parameters, and a time consuming trial and error approach might not lead to significant benefits.

As future work, we plan to extend our system such that real scaling rates are accepted. A potential solution is to combine ReX with a sampling technique in order to handle real scaling rates [4, 8]. Furthermore, we plan to investigate a solution to extrapolate a database to a particular time in future by adapting

[5] cse.ust.hk/graphgen
[6] ibmquestdatagen.sourceforge.net

the existing approach. This represents an interesting future direction, as it raises the challenge of studying an evolving dataset. Moreover, we plan to apply ReX on an existing testing environment from our industrial partner, IBM, and use the extrapolated database for testing the scalability of the system under test.

Acknowledgments. This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie). The authors also acknowledge Dr. Nicola Stokes' feedback.

References

1. A. Arasu, R. Kaushik, and J. Li. Data generation using declarative constraints. In *SIGMOD*, pages 685–696, 2011.
2. C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. Qagen: Generating query-aware test databases. In *SIGMOD*, pages 341–352, 2007.
3. N. Bruno and S. Chaudhuri. Flexible database generators. In *VLDB*, pages 1097–1107, 2005.
4. T. S. Buda, T. Cerqueus, J. Murphy, and M. Kristiansen. CoDS: A representative sampling method for relational databases. In *DEXA*, pages 342–356, 2013.
5. T. S. Buda, T. Cerqueus, J. Murphy, and M. Kristiansen. VFDS: Very fast database sampling system. In *IEEE IRI*, pages 153–160, 2013.
6. D. Chays, J. Shahid, and P. G. Frankl. Query-based test generation for database applications. In *DBTest*, pages 1–6, 2008.
7. Y. Deng, P. Frankl, and D. Chays. Testing database transactions with agenda. In *ICSE*, pages 78–87, 2005.
8. R. Gemulla, P. Rösch, and W. Lehner. Linked bernoulli synopses: Sampling along foreign keys. In *SSDBM*, pages 6–23, 2008.
9. J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. *SIGMOD Record*, 23(2):243–252, 1994.
10. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD*, 11(1):10–18, 2009.
11. J. E. Hoag and C. W. Thompson. A parallel general-purpose synthetic data generator. *SIGMOD Record*, 36(1):19–24, 2007.
12. K. Houkjær, K. Torp, and R. Wind. Simple and realistic data generation. In *VLDB*, pages 1243–1246, 2006.
13. E. Lo, N. Cheng, and W.-K. Hon. Generating databases for query workloads. *PVLDB*, 3(1-2):848–859, 2010.
14. C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *SIGMOD*, pages 245–256, 2009.
15. Y. Pei and O. Zaane. A synthetic data generator for clustering and outlier analysis. Technical report, 2006.
16. G. Ramesh, M. J. Zaki, and W. A. Maniatty. Distribution-based synthetic database generation techniques for itemset mining. In *IDEAS*, pages 307–316, 2005.
17. J. M. Stephens and M. Poess. MUDD: a multidimensional data generator. In *WOSP*, pages 104–109, 2004.
18. K. Taneja, Y. Zhang, and T. Xie. MODA: Automated test generation for database applications via mock objects. In *ASE*, pages 289–292, 2010.
19. Y. Tay, B. T. Dai, D. T. Wang, E. Y. Sun, Y. Lin, and Y. Lin. UpSizeR: Synthetically scaling an empirical relational database. *Information Systems*, 38(8):1168 – 1183, 2013.