A Collaborative Framework for Tweaking Properties in A Synthetic Dataset

J.W. Zhang
National University of Singapore
jiangwei@u.nus.edu

Yu Wang
National University of Singapore
yuwang@u.nus.edu

Y.C. Tay
National University of Singapore
dcstayyc@nus.edu.sg

ABSTRACT

Researchers and developers use benchmarks to compare their algorithms and products. For database systems, a benchmark must have a dataset $\mathcal{D}$. To be application-specific, this dataset $\mathcal{D}$ should be empirical. However, a real $\mathcal{D}$ may be too small, or too large, for the benchmarking experiments. Therefore, $\mathcal{D}$ must first be scaled to the desired size.

Previous related work typically extracts a set of properties $\Pi = \{\pi_1, \ldots, \pi_n\}$ from $\mathcal{D}$, then use Π to generate the synthetic $\widetilde{\mathcal{D}}$. Π may thus ensure $\widetilde{\mathcal{D}}$ is similar to $\mathcal{D}$. This approach of having some monolithic software enforce properties $\pi_1, \ldots, \pi_n$ becomes increasingly intractable as n increases. Our demonstration will present ASPECT, a framework that takes a different approach.

With ASPECT, there is a tool S_0 to first scale the dataset size. The resulting $\widetilde{\mathcal{D}}$ can then be tweaked by tools $\mathcal{T}_1, \ldots, \mathcal{T}_n$, where $\mathcal{T}_k$ enforces π_k in $\widetilde{\mathcal{D}}$.

At the demonstration, a visitor has a choice of (i) $\mathcal{D}$, (ii) size scaler S_0, (iii) the subset of properties to enforce, and (iv) the order of applying the tools for the chosen properties. The visitor can then see the enforcement error for each π_k and the running time for each $\mathcal{T}_k$.

A video of the demonstration is presented here:
http://scaler.d2.comp.nus.edu.sg/

PVLDB Reference Format:
J.W. Zhang, Yu Wang and Y.C. Tay. A Collaborative Framework for Tweaking Properties in a Synthetic Dataset. *PVLDB*, 11 (12): 2010-2013, 2018.
DOI: https://doi.org/10.14778/3229863.3236247

1. INTRODUCTION

Benchmarks are ubiquitous for the computing industry and in academia. Developers and researchers use them to compare their products and algorithms. For 20-odd years, the popular benchmarks for database management systems were the ones defined by the Transaction Processing Council (TPC) [1]. However, the small number of TPC benchmarks are increasingly irrelevant to the myriad of diverse applications, and the TPC standardization process is too slow [5]. This led to a proposal for a paradigm shift, from a top-down design of domain-specific benchmarks by committee consensus, to a bottom-up collaboration to develop tools for application-specific benchmarking [6].

A database benchmark must have a dataset. For the benchmark to be application-specific, it must start with an empirical dataset $\mathcal{D}$. This $\mathcal{D}$ may be too small or too large for the benchmarking experiment, so the first tool to develop would be for scaling $\mathcal{D}$ to a desired size. This motivated the *Dataset Scaling Problem* [6]: Given a dataset $\mathcal{D}$ and a scale factor s, generate a synthetic dataset $\widetilde{\mathcal{D}}$ that is similar to $\mathcal{D}$ but s times its size.

The definition of **similarity** is application-specific, and can be specified by a set of properties $\Pi = \{\pi_1, \ldots, \pi_n\}$. For example, if $\mathcal{D}$ is a social network dataset, π_i may be "5% of users have more than 100 friends", and π_j may be "80% of posts have no comments".

1.1 Current Techniques and Limitations

Previous solutions to the Dataset Scaling Problem [3, 4, 7, 8] proceed as follows:
(Step1) Extract a set of properties $\Pi = \{\pi_1, \ldots, \pi_n\}$ from $\mathcal{D}$. Each π_i is some metric or statistic, like fraction of posts with no comments or joint distribution of foreign key values.
(Step2) Scale each π_i to some target property $\widetilde{\pi_i}$. $\widetilde{\pi_i}$ may be the same as π_i, but this may not be possible; e.g. if π_i is the frequency distribution of foreign key to primary key references, the number of tuples in $\widetilde{\mathcal{D}}$ may prevent $\widetilde{\pi_i}$ from being exactly the same as π_i.
(Step3) Use $\widetilde{\Pi} = \{\widetilde{\pi_1}, \ldots, \widetilde{\pi_n}\}$ to synthesize $\widetilde{\mathcal{D}}$.

This approach has limitations for the developer and user:
(Code Reuse) Suppose a developer has implemented an algorithm $\mathcal{A}_{12}$ to enforce properties $\{\pi_1, \pi_2\}$, and someone else has implemented some $\mathcal{A}_{23}$ to enforce properties $\{\pi_2, \pi_3\}$. Another developer who wants to enforce $\{\pi_1, \pi_2, \pi_3\}$ will have to add π_3 enforcement to $\mathcal{A}_{12}$, or add π_1 enforcement to $\mathcal{A}_{23}$, or implement some $\mathcal{A}_{123}$ from scratch. In each case, there is a reimplementation of code for property enforcement. There is another need for code reuse: One can increase the similarity between $\widetilde{\mathcal{D}}$ and $\mathcal{D}$ by having more properties in Π, but the implementation effort for a monolithic software to enforce more properties is also greater.

[1]http://www.tpc.org/

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236247

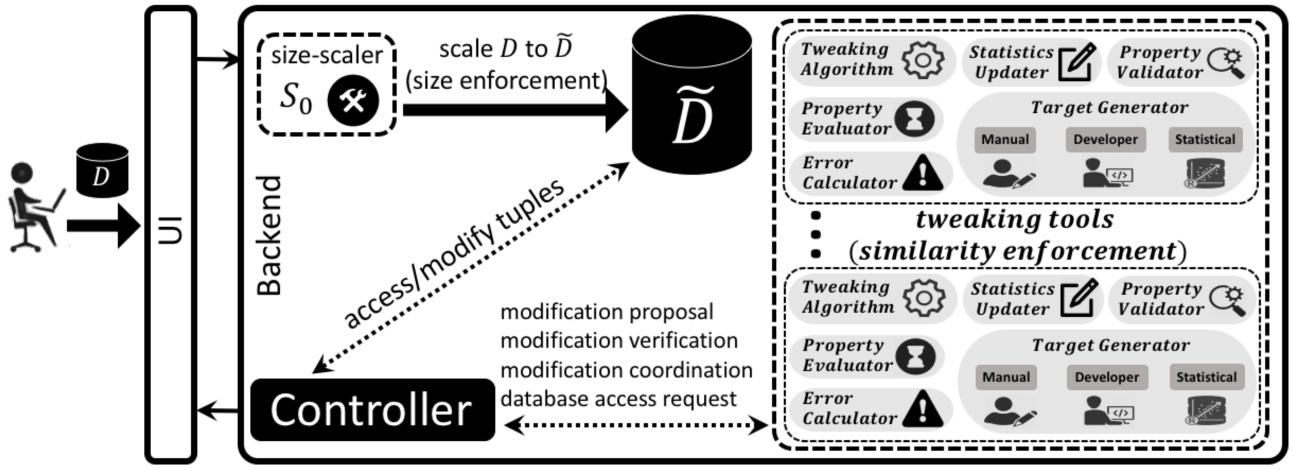


Figure 1: The ASPECT architecture – The empirical dataset $\mathcal{D}$ is uploaded by the user to the backend system through the user interface (UI). Then, $\mathcal{D}$ is first scaled to $\widetilde{\mathcal{D}}$ by a size-scaler S_0 to meet the size requirement. After that, tools $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n$ sequentially interacts with the controller to modify dataset $\widetilde{\mathcal{D}}$ to enforce target properties $\widetilde{\pi_1}, \ldots, \widetilde{\pi_n}$ (similarity enforcement). At the end of modification, the target properties will be reflected in $\widetilde{\mathcal{D}}$.

(Enforcement Flexibility) Given $\mathcal{A}_{12}$ and $\mathcal{A}_{23}$ in the example above, a user cannot freely choose to enforce just π_1, nor change the order in enforcement of π_2 and π_3.

1.2 Our Solution: ASPECT

In this demonstration, we present ASPECT, a framework for scaling an empirical dataset $\mathcal{D}$ to a synthetic $\widetilde{\mathcal{D}}$, and flexibly enforcing similarity between $\mathcal{D}$ and $\widetilde{\mathcal{D}}$. ASPECT has a repository of software tools $S_0, \mathcal{T}_1, \ldots, \mathcal{T}_n$:
(S_0) There is a choice of S_0 for scaling $\mathcal{D}$ to the desired size. This S_0 may enforce similarity for some basic properties, e.g. the number of foreign key references for each primary key value.
($\mathcal{T}_k$) Each $\mathcal{T}_k$ is a tool to enforce some target property $\widetilde{\pi_k}$; specifically, $\mathcal{T}_k$ **tweaks** (i.e. modifies) the data in $\widetilde{\mathcal{D}}$ to enforce $\widetilde{\pi_k}$. To enforce $\widetilde{\pi_1}$ and $\widetilde{\pi_2}$, a user can choose $\mathcal{T}_1(\mathcal{T}_2(\widetilde{\mathcal{D}}))$ or $\mathcal{T}_2(\mathcal{T}_1(\widetilde{\mathcal{D}}))$ by ordering the application of the tools.

The ASPECT approach addresses the above-mentioned limitations as follows:
(Code Reuse) If some developer X has implemented tools $\mathcal{T}_1$ and $\mathcal{T}_2$ to enforce properties $\widetilde{\pi_1}$ and $\widetilde{\pi_2}$, and someone else Y wants to enforce properties $\widetilde{\pi_2}$ and $\widetilde{\pi_3}$, then Y just need to implement a tool $\mathcal{T}_3$ for $\widetilde{\pi_3}$, and use $\mathcal{T}_2(\mathcal{T}_3(\widetilde{\mathcal{D}}))$ or $\mathcal{T}_3(\mathcal{T}_2(\widetilde{\mathcal{D}}))$.
(Enforcement Flexibility) A user can choose the subset of tools available from ASPECT, and choose the order for applying the tools on $\widetilde{\mathcal{D}}$.

Since there are innumerable applications and properties, and the list is evergrowing, we envision having developers from the database community contribute the tools in ASPECT. This would go some way towards realizing the vision of a paradigm shift to a bottom-up collaboration for application-specific benchmarking.

However, for the tools from different developers to interoperate, ASPECT must specify the interfaces and provide a framework for the tools to collaborate. We next describe the architecture to support this.

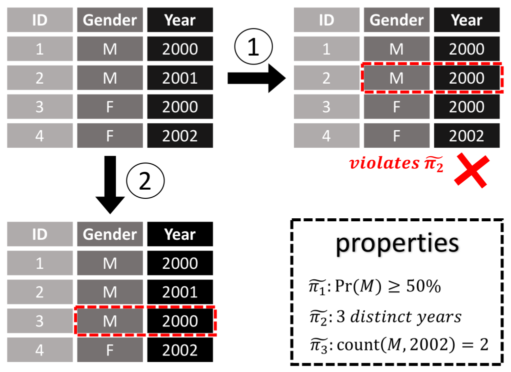

ID
Gender
Year
1
M
2000
2
M
2001
3
F
2000
4
F
2002
1
2
violates $\widetilde{\pi_2}$
properties
$\widetilde{\pi_1}$: Pr(M) ≥ 50%
$\widetilde{\pi_2}$: 3 distinct years
$\widetilde{\pi_3}$: count(M, 2002) = 2

Figure 2: Running example for property enforcement.

2. ASPECT ARCHITECTURE

Figure 1 illustrates the ASPECT architecture. It has a **user interface (UI)** and a **backend system**. We will not elaborate on the UI, which helps the user understand and interact with the backend system. (It is implemented in Java.) Further details can be found in a technical report that is under revision [9].

2.1 The Backend System

The backend system has two tasks: **Size Scaling** and **Property Enforcement**.

2.1.1 Size Scaling

ASPECT first runs a tool S_0 to scale the input empirical $\mathcal{D}$ to a synthetic $\widetilde{\mathcal{D}}$ of the desired size. For example, S_0 may let the user specify the number of suppliers and products in an e-commerce dataset $\widetilde{\mathcal{D}}$. Currently, ASPECT offers three choices for S_0: DSCALER [8], *ReX* [1] and *Rand*(which generates random attribute values for the required number of tuples).

2.1.2 Property Enforcement

We illustrate the property enforcement together with the example presented in Figure 2. In the Figure 2, there are three properties expected to be presented in the scaled database. $\widetilde{\pi_1}$ says that the ratio of male gender should be no less than 50%. $\widetilde{\pi_2}$ says that there must be 3 distinct years. $\widetilde{\pi_3}$ says that there are exactly 2 males born in 2002. Suppose ASPECT has already applied $\mathcal{T}_1, \mathcal{T}_2$ to $\widetilde{\mathcal{D}}$. Now we are going to apply $\mathcal{T}_3$ to fix $\widetilde{\pi_3}$, it does the following:

Step1. *Controller* first calls $\mathcal{T}_3$ to start tweaking, then calls $\mathcal{T}_1, \mathcal{T}_2$ to start preparation.

Step2.1. $\mathcal{T}_3$ calls *Target Generator* to determine $\widetilde{\pi_3}$. In the Figure 2, $\widetilde{\pi_3}$ is labelled accordingly.

Step2.2. Concurrently, $\mathcal{T}_1, \mathcal{T}_2$ call their respective *Property Evaluator* to calculate current statistics for their properties.

Step3. $\mathcal{T}_3$ starts its *Tweaking Algorithm*. Every time $\mathcal{T}_3$ needs to modify $\widetilde{\mathcal{D}}$, it sends the intended modification to *Controller* for validation.

Step4. For each proposed modification, *Controller* calls $\mathcal{T}_1, \mathcal{T}_2$ to get the agreement of their *Property Validators*. *Controller* summarizes the feedback from $\mathcal{T}_1, \mathcal{T}_2$ and replies "yes" or "no" to $\mathcal{T}_3$. In Figure 2, if the proposed modification is modifying the **second** row to $< M, 2000 >$, then, it violates $\widetilde{\pi_2}$. However, if the proposed modification is modifying the **third** row to $< M, 2000 >$, then such a proposal should be accepted.

Step5.1. If the reply is "yes", *Controller* modifies $\widetilde{\mathcal{D}}$ and tells $\mathcal{T}_1, \mathcal{T}_2$ to run their *Statistics Updator*.

Step5.2. If the reply is "no", $\mathcal{T}_3$ must find an alternative modification.

Step6. Repeat from Step3 until $\mathcal{T}_3$ halts.

Of course, there are properties that are mutually conflicting, hence cannot be satisfied at the same time. To resolve this, ASPCET always enforces the properties for the most recently applied tools. Reader can find more details in the technical report [9].

2.2 Tool Implementation Details

To facilitate interoperability, ASPECT requires each tool to have 6 components that satisfy the following specifications:

(1) Target Generator:
This module generates the target property $\widetilde{\pi_k}$ for $\mathcal{T}_k$. For $\widetilde{\mathcal{D}}$ to be similar to $\mathcal{D}$ with respect to π_k, $\widetilde{\pi_k}$ would be the same as π_k, as determined from $\mathcal{D}$. However, ASPECT allows 3 alternative ways of specifying $\widetilde{\pi_k}$:

(1a) User Input The user can accept π_k as the default $\widetilde{\pi_k}$, or specify the target $\widetilde{\pi_k}$ manually. For example, the user may want to specify the fraction of males in $\widetilde{\mathcal{D}}$.

(1b) Developer Generation A developer who implements $\mathcal{T}_k$ may have a better understanding of how π_k (e.g. for the number of comments per post) changes when the dataset scales up or down. The developer may therefore provide the code for generating $\widetilde{\pi_k}$.

(1c) Statistical Extrapolation Some datasets may have a time attribute that can be used to take snapshots $\mathcal{D}_1, \ldots, \mathcal{D}_r$ of $\mathcal{D}$. Otherwise, the user can provide some sampling technique to obtain $\mathcal{D}_1, \ldots, \mathcal{D}_r$ (with increasing size), or accept the default VFDS [2] sampling provided by ASPECT. ASPECT then extracts statistics of the desired property from $\mathcal{D}_1, \ldots, \mathcal{D}_r$, fit these statistics with some distribution, and extrapolate the distribution parameters to get the target property. For example, the number of comments per post in $\mathcal{D}_i$ may be modelled by a Poisson(λ_i) distribution, and $\lambda_1, \ldots, \lambda_r$ fitted by some polynomial that is used to determine the target λ for $\widetilde{\mathcal{D}}$. Currently, ASPECT can do such prediction for statistics in the form of frequency distribution f, where $\sum_v f(v) = 1$ and v is a vector of attribute values (e.g. $<$ `weight`, `height` $>$ or $<$ `age`, `income`, `gender` $>$).

(2) Tweaking Algorithm:
It tweaks the dataset $\widetilde{\mathcal{D}}$ to make sure the target property $\widetilde{\pi_k}$ is enforced in $\widetilde{\mathcal{D}}$ when the algorithm terminates. Note that some properties (e.g. for the fraction of user-pairs who comment on each other's posts) require nontrivial tweaking algorithms.

(3) Property Evaluator:
This calculates statistics for a property.

(4) Property Validator:
It checks whether a proposed tuple insertion/deletion/replacement adversely affects an existing property. Suppose, to enforce a property $\widetilde{\pi_k}$, the tool $\mathcal{T}_k$ proposes to modify a tuple t to become t', but t' will cause a violation of some currently enforced $\widetilde{\pi}_i$. Then the property validator for $\mathcal{T}_i$ will vote against t' and $\mathcal{T}_k$ must find some alternative t''. If no such alternative is possible, ASPECT can allow a modification to proceed, and accept an error increase in property enforcement.

(5) Statistics Updator:
The relevant statistics are updated after each tuple modification.

(6) Error Calculator:
This component calculates the error between the property in the scaled database $\widetilde{\mathcal{D}}$ and the target property.

It is the developer's responsibility to ensure that a tool correctly implements the 6 requirements above, and comply with the ASPECT framework. If a tool does not, say, properly validate a proposed modification, then it will likely fail to enforce its corresponding property.

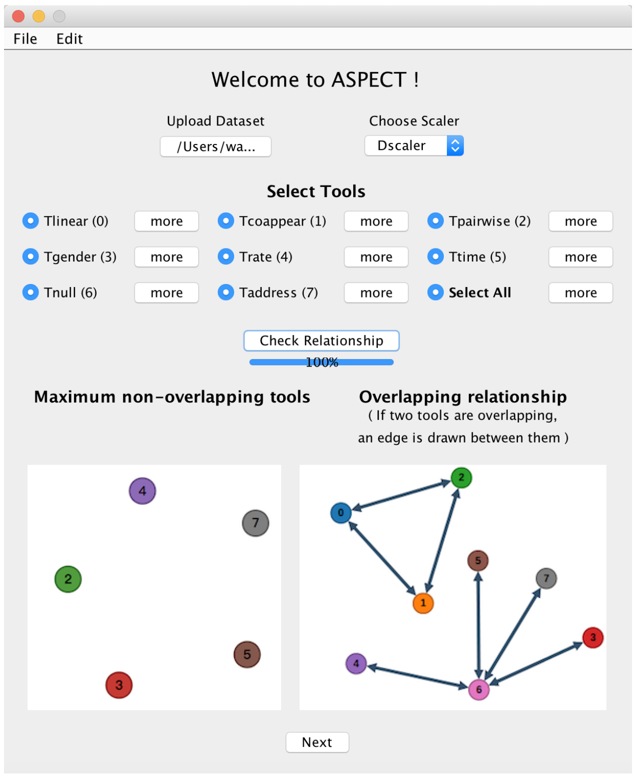
3. DEMONSTRATION SCENARIO

In this demonstration, a visitor first chooses between two (real) social network datasets (`Xiami` [2] and `Douban` [3]), and picks the size scaler S_0 (DSCALER, *ReX* or *Rand*). The UI will list the tweaking tools currently in ASPECT's repository; it also indicates whether two tools *overlap*, in the sense that their properties include common attributes (e.g. "80% of posts have no comments" and "40% of comments are from female users" overlap since they refer to the same attribute `CommentID`). The visitor then selects which tools to apply. These choices are illustrated in Figure 3. Some of the tools might be overlapping. ASPECT will automatically find out those overlapping tools as presented at the bottom of Figure 3.

The visitor then specifies the order for applying the chosen tools. By specifying different execution order of the chosen tools, the visitor can gain some intuition of how the properties are enforced.

Next, ASPECT starts tweaking the dataset. ASPECT applies the tool one by one. Whenever a tool t' will cause a violation of some currently enforced $\widetilde{\pi}_i$, ASPECT will pop out a window to ask the visitor to choose skip validation on conflicting properties. If the visitor does not respond

[2]www.xiami.com
[3]www.douban.com



Welcome to ASPECT !

Upload Dataset: /Users/wa...

Choose Scaler: Dscaler

Select Tools

Tlinear (0), Tcoappear (1), Tpairwise (2), Tgender (3), Trate (4), Ttime (5), Tnull (6), Taddress (7), Select All

Maximum non-overlapping tools

Overlapping relationship (If two tools are overlapping, an edge is drawn between them)

Figure 3: Tool Selection Interface: Each node in the graph is a tool, e.g. $\mathcal{T}_{\texttt{linear}}$ is the node with id 0. An edge between two tools indicate that their properties overlap, i.e. have a common attribute.

within one minute, ASPECT will randomly skip on currently enforced property.

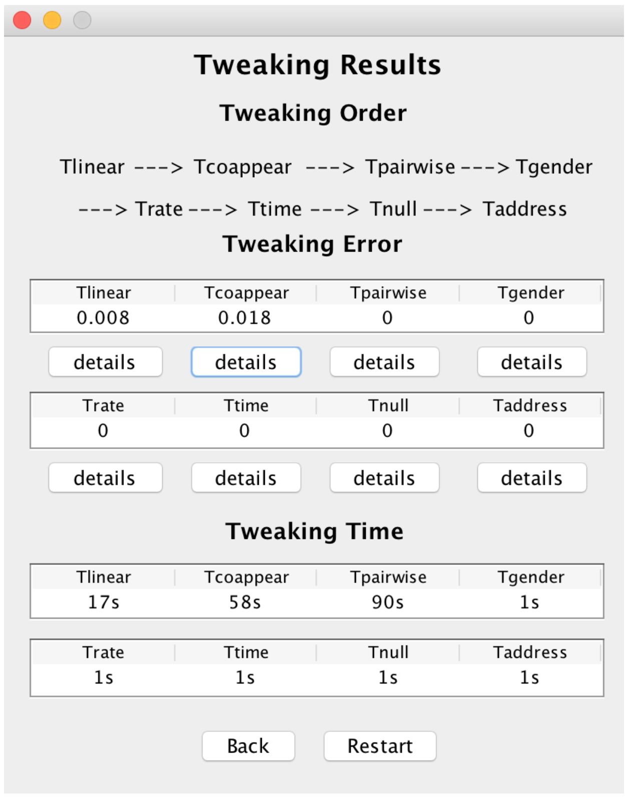
When this is done, ASPECT presents the error rate for each chosen property π_i and the time taken for the corresponding $\mathcal{T}_i$. This is illustrated in Figure 4. The visitor can check how the error of each property is calculated by clicking the `details` button.

Visitors can experiment with the demo by changing their choices to see the impact on $\widetilde{\mathcal{D}}$.

4. CONCLUSION

ASPECT is our contribution towards shifting database benchmarking from TPC-like synthetic datasets to a scale-then-tweak approach that is based on empirical datasets. We will upload the source code of ASPECT to `GitHub`.

The success of this approach relies on collecting a critical mass of tweaking tools from developers and researchers in the database community. This demonstration is therefore an effort to reach out to them, get them acquainted with ASPECT, and build a community of developers for tweaking tool.



Tweaking Results

Tweaking Order

Tlinear ---> Tcoappear ---> Tpairwise ---> Tgender ---> Trate ---> Ttime ---> Tnull ---> Taddress

Tweaking Error

Tlinear	Tcoappear	Tpairwise	Tgender
0.008	0.018	0	0

Trate	Ttime	Tnull	Taddress
0	0	0	0

Tweaking Time

Tlinear	Tcoappear	Tpairwise	Tgender
17s	58s	90s	1s

Trate	Ttime	Tnull	Taddress
1s	1s	1s	1s

Figure 4: Result Interface: If a tweaking process finishes within 1 second, it is indicated as 1s.

5. REFERENCES

[1] T. Buda, T. Cerqueus, et al. ReX: Extrapolating relational data in a representative way. In *Data Science*, LNCS 9147, pages 95–107. Springer, 2015.

[2] T. S. Buda, T. Cerqueus, et al. VFDS: An application to generate fast sample databases. In *CIKM*, pages 2048–2050, 2014.

[3] L. Gu, M. Zhou, Z. Zhang, et al. Chronos: An elastic parallel framework for stream benchmark generation and simulation. In *ICDE*, pages 101–112, 2015.

[4] N. Patki, R. Wedge, and K. Veeramachaneni. The synthetic data vault. In *DSAA*, pages 399–410, Oct 2016.

[5] M. Stonebraker. A new direction for TPC? In *TPCTC*, pages 11–17, 2009.

[6] Y. C. Tay. Data generation for application-specific benchmarking. *PVLDB*, 4(12):1470–1473, 2011.

[7] Y. C. Tay, B. T. Dai, et al. UpSizeR: Synthetically scaling an empirical relational database. *Inf. Syst.*, 38(8):1168–1183, 2013.

[8] J. W. Zhang and Y. C. Tay. Dscaler: Synthetically scaling a given relational database. *PVLDB*, 9(14):1671–1682, 2016.

[9] J. W. Zhang and Y. C. Tay. A tool framework for tweaking features in synthetic datasets. `https://arxiv.org/abs/1801.03645, 2018.`