A tool framework for tweaking features in synthetic datasets

J.W. Zhang #1, Y.C. Tay #2

# School of Computing, National University of Singapore

#1 jiangwei@u.nus.edu

#2 dcsstayyc@nus.edu.sg

Abstract—Researchers and developers use benchmarks to compare their algorithms and products. A database benchmark must have a dataset $D$. To be application-specific, this dataset $D$ should be empirical. However, $D$ may be too small, or too large, for the benchmarking experiments. $D$ must, therefore, be scaled to the desired size.

To ensure the scaled $\tilde{D}$ is similar to $D$, previous work typically specifies or extracts a fixed set of features $F = \{F_1, F_2, \ldots, F_n\}$ from $D$, then uses $F$ to generate synthetic data for $\tilde{D}$. However, this approach $(D \rightarrow F \rightarrow \tilde{D})$ becomes increasingly intractable as $F$ gets larger, so a new solution is necessary.

Different from existing approaches, this paper proposes ASPECT to scale $D$ to enforce similarity. ASPECT first uses a size-scaler $(S_n)$ to scale $D$ to $\tilde{D}$. Then the user selects a set of desired features $\tilde{F}_1, \ldots, \tilde{F}_n$. For each desired feature $\tilde{F}_i$, there is a tweaking tool $T_i$ that tweaks $\tilde{D}$ to make sure $\tilde{D}$ has the required feature $\tilde{F}_i$. ASPECT coordinates the tweaking of $T_1, \ldots, T_n$ to $\tilde{D}$, so $T_n(\cdots(T_1(\tilde{D}))\cdots)$ has the required features $\tilde{F}_1, \ldots, \tilde{F}_n$.

By shifting from $D \rightarrow F \rightarrow \tilde{D}$ to $D \rightarrow \tilde{D} \rightarrow \tilde{F}$, data scaling becomes flexible. The user can customise the scaled dataset with their own interested features. Extensive experiments on real datasets show that ASPECT can enforce similarity in the dataset effectively and efficiently.

I. INTRODUCTION

We have two motivations for introducing ASPECT:

Motivation 1: Benchmarks are ubiquitous in the computing industry and academia. Developers use benchmarks to compare products and algorithms, while researchers use them similarly in research.

For 20-odd years, the popular benchmarks for database management systems were the ones defined by the Transaction Processing Council (TPC). However, the small number of TPC benchmarks are increasingly irrelevant to the myriad of diverse applications, and the TPC standardization process is too slow. 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.

A database benchmark must have a dataset. For the benchmark to be application-specific, it must start with an empirical dataset $D$. This $D$ may be too small or too large for the benchmarking experiment, so the first tool to develop would be for scaling $D$ to a desired size.

Motivation 2: Apart from benchmarking, dataset scaling plays important roles in other fields as well. A start-up company with a small dataset may want a larger dataset for testing the scalability of their system architecture. On the other hand, an enterprise with a large dataset may want a scaled down version to provide quick answers to aggregation queries (averages, count, etc.).

Given this outlook, a tool that scales an empirical dataset $D$ to a synthetic and similar $\tilde{D}$ will be very appealing. This generation of artificial data is necessary if $\tilde{D}$ is larger, and helpful if $\tilde{D}$ is smaller or equal in size. For all cases: $\tilde{D}$ must be similar to $D$. Moreover, the similarity definition should be application-specific. It can be measured by graph properties, query results, etc., depends on the application.

A. Existing approach and the limitations

To ensure $\tilde{D}$ is similar to $D$, previous work typically follows the framework in Fig.1. Each algorithm extracts a fixed set of features $F = \{F_1, F_2, \ldots, F_n\}$ from $D$, then scales $F$ to $\tilde{F}$ as a predicting feature for the scaled dataset $\tilde{D}$. $\tilde{D}$ is then synthesized based on $\tilde{F}$. $\tilde{F}$ here defines the similarity between $D$ and $\tilde{D}$: the more features in $\tilde{F}$, the greater the similarity between $D$ and $\tilde{D}$.

For example, if $D$ is a graph and $F = \{F_1, F_2\}$, where $F_1$ is density and $F_2$ is number of triangles, then we would expect $D$ and $\tilde{D}$ are similar in terms of density and triangles. However, there are some limitations from the perspective of a developer and a user:

1) The developer faces the implementation reusability and scalability issue: Algorithm Implementation Reusability: Consider the scenario where one application developer implements an algorithm $A_1$ using the feature set $\{F_1, F_2\}$. Later, another developer may find it more important for her application to preserve $\{F_2, F_3\}$, where $F_3$ is the number of rectangles. So she implements another algorithm $A_2$ to preserve $\{F_2, F_3\}$. However, a third developer might want to preserve $\{F_1, F_2, F_3\}$: what should he do? In this case,
A₁ or A₂ only preserves part of \{F₁, F₂, F₃\}. To preserve \{F₁, F₂, F₃\}, one has to modify A₁ to preserve the extra \(F₃\), or modify A₂ to preserve the extra \(F₁\), or write a new algorithm A₃ from scratch. This is a waste of effort, since \(F₁, F₂, F₃\) were already preserved by A₁ and A₂. **Algorithm Implementation Scalability:** As mentioned previously, the more features in \(F\), the greater the similarity between \(D\) and \(\bar{D}\). However, a large feature set dramatically increases the difficulty of designing an algorithm that maintains the features simultaneously. For example, if \(F = \{F₁, F₂, F₃, F₄\}\), where \(F₄\) is the fraction of nodes with degree 1, then it is less likely one can design a single algorithm which preserves all 4 features. If we only consider degree distributions as features, then it is already NP-hard to decide whether there exists a graph satisfying certain degree distributions [5].

2) The user does not have a choice of the features: In the current framework, once an algorithm is implemented, the features are fixed. Consider the same example used above, A₁ is implemented to preserve \(F = \{F₁, F₂\}\), and A₂ is implemented to preserve \(F = \{F₂, F₃\}\). The user can only choose to preserve \{F₁, F₂\} or \{F₂, F₃\}, but not the union.

**B. Overcoming the limitations**

In this paper, we propose ASPECT, a flexible framework for synthetic data scaling. Unlike existing approaches, ASPECT takes the following two steps as illustrated in Fig.2:

**Step1:** Use a size-scaler \(S₀\) to scale \(D\) to \(\bar{D}_0\) of desired size.

**Step2:** For the desired feature set \{\(\bar{F}_₁, \bar{F}_₂\),...\(\bar{F}_k\)\}, apply independently developed tools \(T₁, T₂\),...\(Tₖ\) on \(\bar{D}_0\) in order. Each tool \(Tᵢ\) generates a dataset \(\bar{D}_ᵢ\) by adjusting \(\bar{D}_{i-1}\). After the adjustment of \(Tᵢ\), \(\bar{D}_ᵢ\) satisfies \{\(\bar{F}_₁, \bar{F}_₂\),...\(\bar{F}_ᵢ\)\}. Note that Step2 does not depend on the size-scaler in Step1. We call this **tweaking** \(\bar{D}_{i-1}\) by tool \(Tᵢ\). The final dataset is \(\bar{D}\).

For the above-mentioned limitations in Sec.I-A, ASPECT resolves them with ease: For implementation reusability, each feature tweaking tool is independently developed. Once a tweaking tool \(Tᵢ\) for feature \(\bar{F}_ᵢ\) is implemented, then the user can apply \(Tᵢ\) together with other tweaking tools whenever it is needed. No re-coding! For implementation scalability, to preserve the feature set with \(n\) features, the developer just needs to implement \(n\) tweaking tools, instead of hardcoding all \(n\) features into a single piece of software. And each tweaking tool \(Tᵢ\) tweaks the feature \(\bar{F}_ᵢ\). For the issue of **feature choice**, once tweaking tools \(T₁, T₂, T₃\) and \(T₄\) are implemented for \(\bar{F}_₁, \bar{F}_₂, \bar{F}_₃\) and \(\bar{F}_₄\) respectively, the user can choose \(\{T₂, T₃, T₄\}\) to get \(\{\bar{F}_₂, \bar{F}_₃\}\), or \(\{T₁, T₃, T₄\}\) to get \(\{\bar{F}_₁, \bar{F}_₃, \bar{F}_₄\}\), etc.

Hence, to enforce greater similarity in the scaled dataset \(\bar{D}\), we just need to apply more tweaking tools. We envision having developers from the database community contributing tools \(Tᵢ\) to a repository for tweaking synthetic datasets. Then ASPECT will have more tools for the user to customise the scaled datasets. This would go some way towards realising the suggested paradigm shift to a bottom-up collaboration for application-specific benchmarking.

However, the tools require some coordination, since some changes to \(\bar{D}_{i-1}\) by one tool \(Tᵢ\) may be undone by another tool \(Tⱼ\). Moreover, tools are developed independently by different developers. Different developers might have different tweaking techniques. To ensure a tool is compatible with ASPECT, ASPECT must explicitly standardise types of modifications that could be made on a dataset while tweaking.

**C. Overview**

To summarize, our contribution in this paper are fourfold:

1. We propose ASPECT, a framework for flexible application of tweaking tools to enforce target features in synthetic dataset.
2. We present results from extensive experiments on real datasets, to verify that ASPECT can enforce similarity in the dataset effectively and efficiently.
3. We present necessary and sufficient conditions, and tweaking algorithms, for three new complex features.
4. We state Feature Tweaking Bound and Order Problems that offer a rewarding challenge for research on dataset tweaking.

We first introduce ASPECT architecture in Sec.II, followed by three new complex features that serve to illustrate the ASPECT framework in Sec.III. One of them concerns inter-column and inter-row correlation induced by implicit relationships in a social network dataset; we thus provide here a solution to a problem highlighted previously [32]. Sec.IV describes the datasets and similarity measures used in the experiments, and the results are presented in Sec.V. Sec.VI points out some limitations and insights of ASPECT. Related work is surveyed in Sec.VII, before Sec. VIII concludes with a summary.

**II. ASPECT ARCHITECTURE**

As shown in Fig.2, an input dataset is first scaled by the size-scaler \(S₀\) which returns a scaled dataset \(\bar{D}_0\) of the desired size. Note that \(S₀\) could be any tool which guarantees the number of tuples in each table generated is as expected and there are no invalid foreign key values. For example, \(S₀\) may be DSCALER [36], or it could be ReX [8]; we will show that ASPECT is able to preserve the features well for both DSCALER and ReX in Sec.V. The choice of \(S₀\) is outside the scope of this paper. After the dataset is resized, ASPECT then coordinates the application of tools \(Tᵢ\) on \(\bar{D}_{i-1}\), to make sure the feature \(Fᵢ\) is reflected in the tweaked dataset \(\bar{D}_ᵢ = Tᵢ(\bar{D}_{i-1})\). In the tweaking process, there are a few issues:

1. How do we get the target feature \(Fᵢ\)?

![Fig. 2: The ASPECT abstract – The empirical dataset \(D\) is first scaled to \(\bar{D}_0\) by a size-scaler \(S₀\). After that, tools \(T₁, T₂, \ldots, Tₙ\) sequentially tweak dataset for features \(\bar{F}_₁, \bar{F}_₂, \ldots, \bar{F}_ₙ\). After \(Tᵢ\) tweaks \(\bar{D}_{i-1}\), this results in \(\bar{D}_ᵢ\). \(\{\bar{F}_₁, \bar{F}_₂, \ldots, \bar{F}_ᵢ\}\) is reflected in \(\bar{D}_ᵢ\).](image)
12. Given an target feature $F_i$, how can we tweak $\tilde{D}_{i-1}$ to ensure that the tweaked dataset $\tilde{D}_i$ contains $F_i$?

13. Given $\tilde{D}_{n-1}$ already contains $F_1, F_2, \ldots, F_{n-1}$, how can we maintain $F_1, F_2, \ldots, F_{n-1}$ while tweaking $F_n$?

14. Tools are developed independently by different developers. How can we make sure these independently developed tools are compatible with ASPECT?

### A. ASPECT flow

To address the above 4 issues, we illustrate the tweaking process for, say, $T_4$.

**Step1.** ASPECT first calls the tool $T_4$ to start tweaking, then calls previously applied tools $T_1, T_2, T_3$ to start preparation.

**Step2.1.** $T_4$ then finds the target feature $F_4$ by calling its Feature Generator.

**Step2.2.** $T_1, T_2, T_3$ call their respective Feature Calculators to calculate the corresponding features $\tilde{F}_1, \tilde{F}_2, \tilde{F}_3$. This step is concurrent with Step 2.1.

**Step3.** $T_4$ starts the Tweaking Algorithm. Every time $T_4$ needs to modify $\tilde{D}_3$, $T_4$ sends the intended modification to ASPECT for validation.

**Step4.** ASPECT calls $T_1, T_2, T_3$ to confirm the modification with their own Feature Validators. ASPECT summarizes the feedback from $T_1, T_2, T_3$ and replies “yes/no” to $T_4$.

**Step5.1.** If the reply is “yes”, ASPECT modifies $\tilde{D}_3$ and tells $T_1, T_2, T_3$ to update the feature statistics by using their Feature Updaters.

**Step5.2.** If the reply is “no”, ASPECT tells $T_4$ to find the an alternative modification.

**Step6.** Repeat from Step3 until $T_4$ halts.

In the flow above, I1 is addressed by Step2.1 (Feature Generator), I2 is addressed by Step3 (Tweaking Algorithm) and I3 is handled in Step2.2, Step4 and Step5 (Feature Calculator, Feature Validator, Feature Updater). We will explain I1, I2, and I3 in Sec.II-B, and I4 in Sec.II-C in detail.

### B. Tweaking tool component

We now explain how each individual tool $T_i$ should be implemented. Each $T_i$ must have at least 5 components.

**Feature Generator:** This module generates the target feature statistics for the tweaked dataset. Such a generation can be done in 3 ways: (i) **User input:** The user might have their own target feature statistics for the scaled dataset. For example, the user might want to specify the number of males in the population. Hence, the user can manually input target features. (ii) **Developer generation:** When a developer implements the tweaking tool for a specific feature, the developer has a better understanding of how the feature changes while the dataset scales. Therefore, the developer can provide the feature generation tool for his/her own developed feature. (iii) **Generate through historical data:** Apart from the previous two methods, statistical tools can be developed for certain features for general purposes, e.g. frequency distribution for attribute values. One can first take chronological snapshots of the dataset (if applicable), $D_1, D_2, \ldots, D_k$, then extract the feature $F_i$ from each snapshot dataset. Next, apply data fitting methods on $F_i$ to fit $\tilde{F}_i$ into different statistical models, e.g. Poisson distribution. Once the best matching model is learned, say, Poisson distribution, we can learn how the shape parameter $\lambda$ varies as the dataset grows. Hence, we can get the target feature $F_i$. Such an approach is orthogonal to this paper, and will be elaborated in a separate paper.

**Tweaking Algorithm:** It tweaks the dataset $\tilde{D}_{i-1}$ to make sure that $D_i$ has the target feature $F_i$ at the end of tweaking. Note that it is not trivial to provide a tweaking algorithm for a complex feature, e.g. *linear* feature as presented in Sec. III. The developer has to code the tweaking algorithm. Moreover, the tweaking algorithm can only modify the dataset through similar operations presented in Fig. 3

**Feature Calculator:** It calculates the feature statistics for $F_i$ from a given dataset.

**Feature Validator:** It checks whether a proposed tuple insertion/deletion/replacement affects some existing feature. Assuming 1 modification on a tuple is needed when tweaking feature $\tilde{F}_n$ on dataset $\tilde{D}_{n-1}$. Modifying either $t_1$ or $t_2$ will satisfy $\tilde{F}_n$, and modifying $t_1$ changes a previously tweaked feature $\tilde{F}_1$, but modifying $t_2$ does not. Then, modify $t_2$ instead of $t_1$.

At times, it is too strict when a tuple modification is allowed only if no previously tweaked feature affected. For example, $F_1 = \{\text{more than half of the customers are men}\}$ and $F_2 = \{\text{more than half of the customers are women}\}$. These 2 features are contradictory, so one of it has to be violated. Hence, the validation needs to be relaxed. In this paper, a tuple modification is allowed if the resulting errors for all previously tweaked features are less than 5%, which is the threshold $\epsilon_{\text{threshold}}$. Consider the example used previously: if error $< \epsilon_{\text{threshold}}$, then modification on $t_1$ is allowed as well; however, if error $\geq \epsilon_{\text{threshold}}$, then only $t_2$ can be chosen.

In the worst case, if no tuple modification can satisfy previously tweaked feature’s $\epsilon_{\text{threshold}}$, ASPECT allows more tuple modification by relaxing the validation on fewer features. So, some of the feature’s error might be larger than $\epsilon_{\text{threshold}}$.

**Feature Updater:** After each modification of the tuples, the Feature Updater updates the tweaked features’ statistics.

Under ASPECT, it is the tool developers’ responsibility to ensure that the above requirements are correctly implemented and adhere to ASPECT’s structure. If the developer does not, say, properly validate the modifications for Feature Validator, then it is highly likely the corresponding feature will be affected by subsequent tweaks.

### C. ASPECT compatibility guarantee

As mentioned in I4, we need to guarantee each individually developed tool can be used in ASPECT. So, we standardise
interface for the Feature Updater and Feature Validator. By having the same structure, all tweaking tools will be compatible with ASPECT. Fig.3 presents the common functions to be implemented for any Feature Updater. Feature Validator uses a similar interface, so it is omitted here.

We classify three types of modifications that can be made on a table with $n$ tuples and $k$ columns excluding primary key.

**deleteValues**: Given a list of tuples (tupleIDs) in a table (tableID), this operation erases some columns (colIndexes) of these tuples. Note that the deleted entries are temporarily empty, new values will be added back via insertValues. In Fig.4, Step1 is a deleteValues operation. It deletes the first and third column’s values of first and second tuple. The erased entries are empty after Step1.

**insertValues**: Given a table (tableID), a list of values $(v_1,\ldots,v_d)$ (colValues), some columns $(c_1,\ldots,c_d)$, this operation adds this $(v_1,\ldots,v_d)$ into the tuples $t_1,t_2,\ldots,t_m$ (insertingTupleIDs), where $v_1$ is the value for column $c_1$. These values can only be inserted into the empty entries resulting from deleteValues. Moreover, the total number of inserted values is the same as the total number of deleted values. In Fig.4, Step2 is an insertValues operation. It inserts $[4,4]$ into the first and third column of second tuple.

**replaceValues**: Given a table (tableID) and a list of its tuples (replacingTupleIDs), this operation replaces some columns (colIndexes) of these tuples with new values (newValues). All these tuples will have the same values (newValues) for the replaced attributes. replaceValues is different from insertValues where the replacing entries must not be empty entries. In Fig.4, Step3 is a replaceValues operation. It replaces the first, second and third columns of second and third tuples with $[7,7,7]$.

Any feature tweaking tool developer has to implement the functions in Fig.3 to update the corresponding features. The types of modifications in ASPECT are not exhaustive. There can be other types of modifications that one wants to make. The developer should transform other modifications to the three basic modifications in Fig.3. In this case, we sacrifice some accuracy to favour generality for ASPECT and ease of programming for the developers. If one wants to validate a modification on two tables A and B, one can validate it on table A and then validate it on table B, then accept the modifications if both tables’ validations are successful. We might have false-positive/false-negative cases for these types of validations, but we believe the effect is minor. In Sec. III-B, we need to modify $k$ tables simultaneously. To fit into ASPECT, we validate/modify the tables one by one. Experiments in Sec. V show this approach is effective.

III. Example Features For Demonstration

In ASPECT, the more tools we apply, the more features we can preserve. As pointed out previously, for some feature $F_j$, it is inevitable that some previously tweaked feature $F_i$ may be affected when tweaking $F_j$. The concern is how much $F_i$ is affected while tweaking $F_j$ under ASPECT.

In this paper, we run experiments to demonstrate how $F_i$ is affected while tweaking $F_j$ empirically. We will apply 3 tools $T_1,T_2,T_3$ sequentially on $D_0$ to fix the features $F_1,F_2,F_3$. At the end of the tweaking, we will examine how well the three features are preserved. We propose three important and complex features: linear, coappear, pairwise as example features. These features are selected based on two criteria:

**Popularity**: Our ultimate goal is to build an application-specific system for dataset scaling. Hence, it only makes sense if the features are widely used. The features we consider are used widely in the literature [1], [14], [18], [29].

**Complexity**: Since we want to check how well ASPECT can maintain features if tools can undo previously applied tools. Hence, the features we use for demonstration should be complex and affect each other. Simple features such as “# of null values in each table”, “# of tuples in each table” are easy to tweak. To avoid presenting a strawman test, we skip such simple features in this paper. Nevertheless, these simple feature tweaking tools are already implemented in ASPECT. The user can apply such simple features if needed.

To the best of our knowledge, we are the first to publish these 3 features. Moreover, we are the first to preserve these 3 features within one dataset.

In this section, we make the following assumption to avoid confusion. If $T'$ references $T$ (denoted $T' \rightarrow T$), it does so via one foreign key constraint only. This assumption can be easily relaxed. Also, $t_1 \rightarrow t_2,t_1 \in T_1,t_2 \in T_2$ means tuple $t_1$ references $t_2$.

A. Linear feature

Applications are often interested in computing $T_k \bowtie \cdots \bowtie T_1$ for some reference chain $T_k \rightarrow \cdots \rightarrow T_1$. For example, to count the number of (distinct) movies with reviews that are commented on by users, one may need to take the join of a reference chain from comments to reviews to movies. This is what we call a linear feature.

Fig.5 illustrates the concept of a linear feature. For any reference chain $T_k \rightarrow \cdots \rightarrow T_1$, the linear feature describes how one tuple $t_i \in T_i$ is transitively referenced by other tuples $t_j \in T_j$, for any $j > i$. In Fig.5, $a_2 \in T_A$ is directly referenced by $b_2,b_3 \in T_B$, and indirectly referenced by $c_1,c_2,c_3 \in T_C$. However, $a_2$ is not indirectly referenced by any tuple in $T_D$. Variants of linear joins are widely used to generate query result approximations, such as database sampling [14] and
Theorem 1: [necessity] Let $H$ be the linear join matrix of $T_k \rightarrow \cdots \rightarrow T_1$ before tweaked by $T_{\text{linear}}$, and $H$ be the target linear join matrix. $H$ can be tweaked to $H'$ only if

$$
H' = \begin{bmatrix}
0 & 0 & \cdots & 0 \\
0 & H_{2,1} & 0 & \cdots \\
\vdots & \vdots & \ddots & \vdots \\
0 & \cdots & \cdots & H_{k,k-1}
\end{bmatrix}
$$

(L1) $h_{j,i} \leq \min_{k \leq i < j} |T_n|$ for all $1 \leq i < j \leq k$.

(L2) $h_{i+1,i} \geq h_{i+2,i} \geq \cdots \geq h_{k,i}$ for all $1 \leq i \leq k - 1$.

(L3) $h_{j,i} \leq h_{j,2} \leq \cdots \leq h_{j,i-1}$ for all $2 \leq j \leq k$.

(L4) $h_{j,i+1} - h_{j,i+1,i+1} \geq h_{j,i} - h_{j,i+1,i}$ for all $1 \leq i < j-1 < k-1$.

Next, we describe how $T_{\text{linear}}$ tweaks $H$ to $H'$. $T_{\text{linear}}$ tweaks $H$ row by row. For ith row, $T_{\text{linear}}$ then tweaks the row entry by entry. $T_{\text{linear}}$ first does leadingAdjust, tweaking $(h_{1,1}, h_{1,2}, \ldots, h_{i,i-1})$ to $(\tilde{h}_{1,1}, \ldots)$, $T_{\text{linear}}$ then does nonLeadingAdjust: it tweaks $(\tilde{h}_{i,1}, \ldots)$ to $(\tilde{h}_{i,1}, \tilde{h}_{i,2}, \ldots) \rightarrow (\tilde{h}_{i,1}, \tilde{h}_{i,2}, \tilde{h}_{i,3}, \ldots) \rightarrow \cdots \rightarrow (\tilde{h}_{i,1}, \tilde{h}_{i,2}, \tilde{h}_{i,3}, \ldots, h_{i,i-1})$. Instead of providing a formal proof, we present an example of tweaking from $H_{eg}$ to $H_{exp}$ in Fig.6 and attach the proofs in the appendix.

Second Row: We are expecting one less root for $T_B \rightarrow T_A$. $T_{\text{linear}}$ chooses an existing root, say $a_1$, and plucks all its descendants ($b_1$) and attach them to some other root, say $a_2$. After such modification, we will have 2 roots $a_2, a_3$ for $T_B \rightarrow T_A$. This is reflected in step 1.

Third Row: No modifications are needed for the first entry. For the second entry, one more root for $T_C \rightarrow T_B$ is expected. Hence, we pluck $c_1$ from $b_2$ and attach $c_1$ to $b_1$. This completes the tweaking for the second row and it is reflected in step 2.

Fig. 6: Tweaking demonstration for $T_{\text{linear}}$. The red dotted lines are the modifications made on the dataset. The linear join matrix after each step is presented at the bottom.
Fig. 7: Coappear feature. There are 3 tables $T_A, T_B, T_C$ referencing to the same tables $T_K, T_H$. $(k_1, h_2)$ appeared together in $T_A, T_B$ for 3 times, in $T_C$ for 1 time.

Fourth Row: For the first entry, we expect 1 more root for $T_D \rightarrow T_C \rightarrow T_B \rightarrow T_A$, say $a_2$. Hence, we pluck $d_1$ from $c_4$ and attach $d_1$ to $c_1$. Now we have 2 roots $a_2, a_3$ for $T_D \rightarrow T_C \rightarrow T_B \rightarrow T_A$, and the last row becomes $(2, 3, 3, 0)$. This is reflected in step 3. For the second entry, we expect 2 more roots for $T_D \rightarrow T_C \rightarrow T_B$. Hence, we pluck $d_2$ from $c_4$ and attach it to $c_2$. This is reflected in step 4, and the last row is $(2, 4, 4, 0)$ now. Lastly, we pluck $d_3$ from $c_5$ and attach it to $c_3$ which ends the tweaking. This is reflected in step 5.

When tweaking the $i$th row, $T_{\text{linear}}$ always plucks the tuples in $ith$ table and attaches them to the $(i-1)$th table. $T_{\text{linear}}$ never re-modify the entries in previously tweaked rows, which gives some intuition that the tweaking is always possible.

So far, we only considered tweaking one linear join matrix. In general, a dataset can have multiple overlapping reference chains. Suppose we have already tweaked the matrix for $T_3 \rightarrow T_3 \rightarrow T_1$, then tweak the matrix for an overlapping $T_4 \rightarrow T_3 \rightarrow T_2$. This can undo the tweaking for $T_4 \rightarrow T_3 \rightarrow T_1$.

The issue is not just for overlapping linear chains but, in general, applies to any pair of tweaking algorithms. For example, running $T''$ (e.g. $T_{\text{pairs}}$) after $T'$ (e.g. $T_{\text{coappear}}$) can undo the work done by $T'$. We do not have a solution yet. Instead, we adopt the heuristic as presented in Sec. II-B.

B. Coappear feature

Fig. 7 illustrates the concept of a coappear feature. The tables $T_A, T_B$ and $T_C$ may be for comment, share and like in a social network service, referencing tables $T_K$ and $T_H$ for post and users. Thus, the same $(\text{postID}, \text{userID})$ may appear multiple times in the same table and in multiple tables. This coappear feature can be used for user profiling: e.g. if Alice comments, shares and likes a post about volunteering many times, it is more likely that Alice is interested in volunteer work [29]. Other examples include group theme prediction [10] and on-line recommendation [18].

Tweaking is done via tweaking a frequency distribution that captures the correlation in foreign key appearances:

Definition 3: Suppose $T_1, \ldots, T_k$ reference the same tables $T_1', \ldots, T_m'$ and $b_1, \ldots, b_m$ coappear as foreign keys $v_i$ times in $T_1', \ldots, v_k$ times in $T_k$. If there are $n$ such $(b_1, \ldots, b_m)$, then $\xi_{T_1', \ldots, T_k}(v_1, \ldots, v_k) = n$. We call $(v_1, \ldots, v_k)$ a coappear vector and $\xi_{T_1', \ldots, T_k}$ the coappear distribution. To simplify notation, we refer to $\xi_{T_1', \ldots, T_k}$ as $\xi$ if there is no ambiguity.

In Fig. 7, $(k_1, h_2)$ appears 3 times in $T_A$, 3 times in $T_B$, and 1 time in $T_C$, so $\xi(3, 3, 1) = 1$. Further, $(k_2, h_3)$ and $(k_3, h_1)$ each appears 1 time in $T_A$, 1 time in $T_B$ and 2 times in $T_C$, so $\xi(1, 1, 2) = 2$.

Like for linear joins, we present necessary and sufficient conditions for tweaking the coappear distribution:

**Theorem 2:** [necessity] Suppose tables $T_1, \ldots, T_k$ reference the same tables $T_1', \ldots, T_m'$ in some $D_i$, $\xi$ is the coappear distribution before being tweaked by $T_{\text{coappear}}$, and $\xi'$ is the target coappear distribution. $\xi$ can be tweaked to $\xi'$ only if:

$$\sum_v v_i \xi'(v) = |T_i| \text{ for } 1 \leq i \leq k$$

$$\sum \xi'(v) = \prod_{i=1}^m |T'_i|.$$  

**Proof:** $(C1)$ $\xi'(v)$ is the number of different foreign key tuples $(b_1, \ldots, b_m)$ with coappear vector $v$. Hence, each $(b_1, \ldots, b_m)$ appears $v_i$ times in $T_i$, so $\sum_v v_i \xi(v) = |T_i|$. $(C2)$ $\sum \xi(v)$ is the total number of different $(b_1, \ldots, b_m)$ foreign key combinations. Since each $b_i$ is unique in $T'_i$, the total number of combinations is $\prod_{i=1}^m |T'_i|$.

Next, we explain how $T_{\text{coappear}}$ tweaks $\xi$ to $\xi'$. Let $\xi^* = \xi - \xi'$. $\Delta^+ = \{|v|\xi^*(v) > 0\}$, $\Delta^0 = \{|v|\xi^*(v) = 0\}$ and $\Delta^- = \{|v|\xi^*(v) < 0\}$. $T_{\text{coappear}}$ works as follows:

For each $v = (v_1, \ldots, v_k) \in \Delta^+$, it adds $|\xi^*(v)|$ more foreign key tuples $(b_1, \ldots, b_m)$, each appearing $v_i$ times in $T_i$.

It does this by looping $|\xi^*(v)|$ times, and in each iteration:

CoappearVectorRetrieve: Pick the closest coappear vector $v' = (v'_1, \ldots, v'_k) \in \Delta^+$, using Manhattan distance.

TupleRetrieve: There may be multiple $b = (b_1, \ldots, b_m)$ foreign key tuples with coappear vector $v'$ (e.g. in Fig. 7, $(k_2, h_3)$ and $(k_3, h_3)$ both have $v' = (1, 1, 2)$). For each $v'$, choose one such $b$.

Tupple Modification: Tuples are tweaked as follows: For $1 \leq i \leq k$, if $v'_i - v_i > 0$, remove $v_i - v'_i$ tuples with foreign key values $b$ from $T_i$; if $v'_i - v_i < 0$, add $v_i - v'_i$ tuples with foreign key values $b$ into $T_i$.

StatsUpdate: Update $\xi^*(v)$ by $1$ and $\xi^*(v')$ by $-1$.

The job is done when the loop terminates. We can prove that the necessary conditions are sufficient for the tweaking.

**Theorem 3:** [sufficiency] Suppose tables $T_1, \ldots, T_k$ reference the same tables $T_1', \ldots, T_m'$. Let $\xi$ be the coappear distribution before tweaking and $\xi'$ the target coappear distribution. If $\xi$ satisfies the necessary conditions in Theorem 2, then $T_{\text{coappear}}$ tweaks $\xi$ to become $\xi'$.
Fig. 8: Pairwise feature. There are two users, u1 has 2 posts p1, p2, and u2 has 1 post p3. Moreover, u1 has 2 response2post on u2’s post. Similarly, u2 has 4 response2post on u1’s post.

and $\xi_{TD,T_E}: T_{coappear}$ only modifies the referencing tables; e.g. tweaking $\xi_{TD,T_E}$ only modifies $T_D$ and $T_E$, without affecting $\xi_{T_G,T_H}, T_G$ and $T_H$. We can thus tweak coappear distributions without affecting each other.

C. Pairwise feature

Fig. 8 illustrates the concept of a pairwise feature. In social networks, the social tie between users u1 and u2 may be implicit, instead of explicitly declared (as friends, say). For example, u1 may respond twice (r1, r2) to a post p3 by u2, whereas u2 responds 4 times (r3, r4, r5, r6) to 2 posts p1 and p2 by u1. Such a feature highlighted by previous work[32] involves both inter-column and inter-row correlation.

For expository convenience, we use sonSchema, a generic database schema for social networks [6]. We focus on 3 tables: users, post, response2post. Each of these can have multiple instantiations: user can represent a company, an advertiser, etc., but we will only consider human users; post tables may record blogs, videos, etc. contributed by user; and response2post may be a share, like, etc. For each type of response2post table, the implicit user-to-user tie is captured by the following distribution:

Definition 4: Let $R$ be a response2post table, and $x, y \in \{0, 1, 2, \ldots\}$. Suppose there are k user pairs $\langle u_1, v_1 \rangle, \ldots, \langle u_k, v_k \rangle$, where $u_i$ responds $x$ times to $v_i$’s post, and $v_i$ responds $y$ times to $u_i$’s post. We denote this as $\rho_R(x, y) = k$, and call $\rho_R$ the pairwise distribution.

Like for $T_{1inear}$, we state necessary and sufficient conditions for tweaking pairwise distributions. To simplify the presentation, this section assumes a user never respond to his/her own post. The appendix relaxes this assumption.

Theorem 4: [necessity] For a response2post table $R$ in some $D_i$, $\rho_R$ is the pairwise distribution before being tweaked by $T_{pairwise}$, and $\rho_R$ is the target pairwise distribution. $\rho_R$ can be tweaked to become $\rho_R$ only if:

(P1) $\bar{\rho}_R(x, y) = k$ means there are $k$ user pairs $\langle u_i, v_i \rangle$, where $u_i$ responds $x$ times to $v_i$’s post and $v_i$ responds $y$ times to $u_i$’s post. This yields $x + y$ tuples in response2post for each $\langle u_i, v_i \rangle$. By symmetry, these $x + y$ tuples also represent $k$ $\langle u_i, u_i \rangle$ pairs, so $\bar{\rho}_R(y, x) = k$.

(P2) As above, for each $\bar{\rho}_R(x, y) = k$, there are $k$ $\langle u_i, v_i \rangle$ pairs, and each pair has $x + y$ tuples in response2post. These $k(x + y)$ tuples are double-counted by $\bar{\rho}_R(x, y)$, so we get the equality in (P2).

(P3) Similarly, there are $|U|(|U| - 1)$ user pairs, and each is counted once by $\bar{\rho}_R(x, y)$, so (P3) follows.

Next, we explain how $T_{pairwise}$ tweaks $\rho$ to $\bar{\rho}$. Let $\rho^*_R = \rho_R - \rho_R$, $\Theta^+ = \{(x,y)|\rho_R^+(x,y) > 0\}$ and $\Theta^- = \{(x,y)|\rho_R^-(x,y) < 0\}$. $T_{pairwise}$ loops through each response2post table $R$. For each $(x, y) \in \Theta^-$, it adds $|\rho^*_R(x, y)|$ pairs $\langle u_i, v_i \rangle$, where user $u_i$ has $x$ and $y$ response2post tuples in $R$ referencing $v_i$’s post. It does this by looping $|\rho^*_R(x, y)|$ times, and in each iteration:

PairwiseVectorRetrieve: Pick $\ell' = (x', y') \in \Theta^+$ that is closest to $(x, y)$ by Manhattan distance.

TupleModification: Choose users $u_i$ and $v_i$ with pairwise vector $(x', y')$ and tweak $u_i$’s responses to $v_i$’s post, as follows: If $x < x'$, then $u_i$ has $x' - x$ more responses to $v_i$’s post than desired, so $T_{pairwise}$ randomly chooses and removes $x' - x$ such responses. If $x > x'$, we add $x - x'$ responses from $u_i$ on $v_i$’s post. If $v_i$ has no post, we artificially create a post for $v_i$. To do this, we pick another user $w_i$ who has more than 1 post and pick a post $p_w$ with minimum responses among $w_i$’s posts; we make $p_w$ a post by $v_i$, and shift the responses to $p_w$ to other posts by $w_i$. If (rare case) all users have at most 1 post, we will make a new post $p$ for $v_i$, and add $x - x'$ responses to $p$. We similarly tweak $v_i$’s responses to $u_i$’s post.

StatsUpdate: Increase $\rho^*_R(x, y)$ and $\rho^*_R(y, x)$ by 1 and decrease $\rho^*_R(x', y')$ and $\rho^*_R(y', x')$ by 1.

We can prove the above mentioned conditions in Theorem 4 are sufficient for $T_{pairwise}$ tweaks $\rho_R$ to $\bar{\rho}_R$ by Theorem 5.

Theorem 5: [sufficiency] For each response2post table $R$ in some $D_i, \rho_R$ is the pairwise distribution before tweaking and $\bar{\rho}_R$ is the target pairwise distribution. If $\bar{\rho}_R$ satisfies the necessary conditions in Theorem 4, then $T_{pairwise}$ tweaks $\rho_R$ to $\bar{\rho}_R$. Moreover, the extra tuples added to the post table $P$ is at most $|U| - |P|$, where $U$ is the user table.

The formal proof is provided in the appendix. Since response2post can have several instantiations (e.g. share, like, etc.), a social network dataset can have multiple pairwise distributions, but they can be tweaked independently. For example, suppose a post table $P$ has two response2post tables $R_1$ and $R_2$, and $\rho_{R_1}$ is tweaked to $\bar{\rho}_{R_1}$ first. When tweaking $\rho_{R_2}$, we only modify the tuples in $R_2$, so it does not affect the tweaked $\bar{\rho}_{R_1}$. Moreover, adding tuples in $P$ does not affect $\bar{\rho}_{R_1}$ as well.

For the above mentioned tweaking tools, they modify the
dataset by calling the functions in Sec.II-C. $T_{linear}$ modifies the dataset through the operation replaceValues, $T_{coappear}$ and $T_{pairwise}$ modify the dataset through the operations deleteValues, insertValues.

IV. EXPERIMENT SETUP

In the following, each experiment is run on a Linux machine with 64GB memory and an Intel Xeon 2.4GHz processor. We now describe the datasets and similarity measures used in our experiments. ASPECT is implemented in Java.

A. Datasets

In this paper, due to the space constraint, we only present experiments on Xiami\(^2\). Xiami contains music-related data with 28 tables and more than 90M tuples. Reader can refer to the appendix for experiments on three more datasets, DoubanBook, DoubanMusic and DoubanMovie. Each dataset is larger than 10GB originally. However, there are columns, e.g. song_name, movie_name, that are irrelevant to the experiments. We do not want to exaggerate ASPECT’s capability of handling big datasets. We hence purposely filter out those irrelevant columns and only conduct experiment on the relevant columns.

We take 6 snapshots of each dataset, $D_1 \subset D_2 \subset D_3 \subset D_4 \subset D_5 \subset D_6$. For each $D_i$, ASPECT takes $D_1$ as input, and first uses a size-scaler to scale $D_1$ to $D_0$, where $D_0$ and $D_i$ are of the same size. We then apply tweaking tools on $D_0$ to achieve the target features. After the tweaking process is done, ASPECT outputs $\tilde{D}_i$ which is similar to $D_i$. For our experiments, we use $D_i$ as the ground-truth, and compare the similarity between $\tilde{D}_i$ and $D_i$.

B. Size-scaler

Size-scalers are orthogonal to enforcing features in the final dataset $\tilde{D}$. Our experiments show that ASPECT is able to generate datasets with small errors for three different size-scalers, DSCALER [36], ReX [8] and Rand, described below:

**DSCALER** is the first solution to scale relational tables by different ratios. It uses a correlation database which captures fine-grained, per-tuple correlations to scale the original dataset.

**ReX** is an automated representative extrapolation technique [8]. It scales all tables by the same ratio. Since tables in the ground truth dataset do not scale uniformly, the targeted features do not satisfy the necessary conditions in Sec.III for datasets generated by ReX. We, therefore, modify the targeted features to enforce the necessary conditions before tweaking.

**Rand** is a randomised size-scaler. The tuples are generated randomly. However, it satisfies two requirements: (i) the number of tuples are generated as expected and (ii) the tuples generated satisfy the foreign key constraints.

C. Similarity measure

As stated previously, the similarity between tweaked dataset $\tilde{D}$ and ground truth dataset $D$ are defined through the feature set $F$. Hence, we measure how well ASPECT preserves the features. In the experiment, we only apply 3 tweaking tools presented in Sec.III to preserve the corresponding features. Hence, we measure similarity based on these 3 features.

1) Feature Accuracy: we individually measure the similarity of the 3 features.

**Linear Feature:** For a target linear join matrix $H$ (ground truth) and the corresponding $\bar{H}$ in the final tweaked dataset, let $\epsilon_H$ be the mean relative error among the entries. For example,

$$\epsilon_H = \frac{1}{N_{FK}} \sum_v ||\xi(v) - \bar{\xi}(v)||,$$

where $N_{FK}$ is the number of foreign key vectors. $\epsilon_H$ is bounded by $\frac{1}{N_{FK}} \sum_v ||\xi(v)|| + ||\bar{\xi}(v)|| = 2$. The linear feature error of $\tilde{D}$ is the mean of all $\epsilon_H$, thus unbounded.

**Coappear Feature:** For each coappear distribution, let $\xi$ be the target (ground truth) and $\bar{\xi}$ the tweaked distribution. The coappear distribution error $\epsilon_{\xi}$ is

$$\epsilon_{\xi} = \frac{1}{N_{FK}} \sum_v ||\xi(v) - \bar{\xi}(v)||,$$

where $N_{FK}$ is the number of foreign key vectors. $\epsilon_{\xi}$ is bounded by $\frac{1}{N_{FK}} \sum_v ||\xi(v)|| + ||\bar{\xi}(v)|| = 2$. The coappear feature error of $\tilde{D}$ is the mean of all $\epsilon_{\xi}$.

**Pairwise Feature:** Similarly, for a pairwise distribution, let $\rho$ be the target (ground truth) and $\bar{\rho}$ the tweaked distribution. The pairwise distribution error $\epsilon_{\rho}$ is

$$\epsilon_{\rho} = \frac{1}{N_{user-pair}} \sum_v ||\rho(v) - \bar{\rho}(v)||,$$

where $N_{user-pair}$ is the number of user pairs; $\epsilon_{\rho}$ is at most 2. The pairwise distribution error of $\tilde{D}$ is the mean of all $\epsilon_{\rho}$.

2) Query Accuracy: we also measure similarity by the result of an aggregate queries (COUNT, AVERAGE) that are related to the 3 features. The query error is measured by $\epsilon_q = \frac{|q(D) - q(\tilde{D})|}{q(D)}$.

V. RESULTS AND ANALYSIS

In our experiment, ASPECT coordinates $T_{linear}$, $T_{coappear}$ and $T_{pairwise}$ on the scaled dataset generated by a size-scaler to realize the corresponding features. There are $3! = 6$ ways of ordering these tweaking tools. We use P-L-C, say, to denote the permutation where $T_{pairwise}$, $T_{linear}$ and $T_{coappear}$ are applied in that order.

We first compare the feature similarity in Sec.V-A, followed by query similarity experiments in Sec.V-B. Later, we discuss the possible improvements in Sec.V-B. Lastly, we present the execution time of ASPECT in Sec.V-D.

A. Feature similarity

For each feature, the plots are organized as follows: x-axis represents the dataset snapshots; y-axis is the feature error. In each plot, we compare how the 6 permutations perform against the baseline (without tweaking).

\(^2\)https://www.xiami.com
In Xiami, there are in total 38 linear join matrices. In general, the later $T_{\text{linear}}$ is applied, the smaller the linear feature error, i.e. C-L-P and P-L-C have smaller errors than L-C-P and L-P-C, and C-P-L and P-C-L have 0 error. All permutations reduce the error tremendously for all size-scalers on all datasets.

Different size-scalers generate a scaled dataset with different errors. In Fig.9, take D6 for example, Dscaler generates a dataset with error around 1.0, while ReX generates a dataset with error around 25. Regardless of the initial error difference, ASPECT is able to reduce the error tremendously after applying the tweaking tools.

There are 12 coappear distributions for Xiami. Fig.10 plots the average error of these coappear distributions. It shows that, like for $T_{\text{linear}}$, the later $T_{\text{coappear}}$ is applied in the tweaking order, the smaller the coappear error. In general, we find that permutations where $T_{\text{coappear}}$ is after $T_{\text{linear}}$ reduces the errors more than if $T_{\text{coappear}}$ is before $T_{\text{linear}}$. This is expected, since $T_{\text{linear}}$ modifies the coappearing tables massively after $T_{\text{coappear}}$ is done.

Similar to linear feature, most tweaking permutations significantly reduce the coappear errors. However, for the plot Dscaler-Xiami, we observe that the tweaking permutation C-L-P and C-P-L have a smaller error reduction. One possible reason may be small original error, that gives limited room for improvement. The other possible reason could be the highly overlapping structure: the coappear distribution involves many tables. Take $\xi_T$ for example, where $T$ is (Listen_Artist, Lib_Artist, Artist_Fan, Artist_Comment). This $\xi_T$ overlaps with 8 linear joins, so it is modified by 8 linear tweaking tools if $T_{\text{linear}}$ is applied after $T_{\text{coappear}}$. This increases the difficulty of getting a validated modification as described in Sec.II-B. We will discuss how to improve the similarity for such highly overlapping features in Sec.V-C.

Nevertheless, ReX-Xiami and Rand-Xiami still have small errors despite such a highly overlapping features.

3) **Pairwise feature**: Xiami has 4 pairwise distributions. Fig.11 plots the error of these pairwise distributions. It again shows that, the later $T_{\text{pairwise}}$ is applied in a tweaking order, the smaller the pairwise feature error in the tweaked dataset. Moreover, all tweaking permutations reduce the errors tremendously for all size-scalers.

In summary, the later a tool $T_i$ is applied, the smaller the error for the feature $F_i$. Moreover, all tweaking permutations reduce the errors tremendously for most of the cases. If the features are highly overlapping, it is possible that the error reduction is not very significant. In the next section, we will discuss how to improve this.

**B. Query similarity**

For each query $q$, we compare the query results on the ground-truth dataset $q(D)$ and the scaled dataset $q(D')$. As mentioned in Sec.IV-B, ReX cannot scale the dataset to arbitrary sizes, so it cannot be used for query similarity experiments. The 4 queries used are: $Q_1$ computes the number of users who have uploaded a photo with commenters; $Q_2$ computes the number of Music Videos that have been commented on by at most 10 different users; $Q_3$ computes the average number of listeners per song; $Q_4$ computes the number of user pairs having interactions through profile page.

Fig.12 presents results for the 4 queries. The first row uses Dscaler as a size-scaler, the second row uses Rand as a size-scaler. The x-axis represents the dataset snapshots, and y-axis represents query error.

As we can see from Fig.12, all tweaking permutations reduce the query error significantly on both size-scalers. The errors are reduced to $< 0.05$ for most of the tweaking.

Nevertheless, ReX-Xiami and Rand-Xiami still have small errors despite such a highly overlapping features.
permutations. For $Q_2$, even though the initial error after the size-scaler is relatively low for $D_2$, ASPECT is still able to reduce the error further.

C. Similarity improvement over iterations

Even though ASPECT significantly reduces the errors for most of the cases, there are some rare exceptions, e.g. Dscaler-Xiami in Fig.10, where ASPECT generates a dataset with coappear error > 0.1. Such cases happen when tools modify previously tweaked features. To improve the performance, we run ASPECT for multiple iterations. Previously, we applied tools $T_{\text{coappear}}, T_{\text{linear}}, T_{\text{pairwise}}$ sequentially for the permutation C-L-P which results in $D$. But now, we apply tools $T_{\text{coappear}}, T_{\text{linear}}, T_{\text{pairwise}}$ on $D$ in the same order with another few iterations. We find that by having more iterations of tweaking, the error is further reduced tremendously.

In Dscaler-Xiami, C-L-P, C-P-L have larger errors. Fig.13 presents the results of C-L-P and C-P-L with more iterations. The x-axis is the features; the y-axis represents the errors; the bar represents the iterations.

For C-L-P, the coappear error is 0.08 for the first iteration, and reduced to 0.04 in the second iteration and further reduced to 0.02 in the third iteration. For linear feature, the error reduction is greater, from 0.05 to $10^{-3}$ from third iteration onwards. For C-P-L, we observe similar phenomenon. Moreover, the error reduction is faster (the error stabilises from second iteration onwards).

In summary, the errors are reduced significantly as the number of iterations increases. From the second or third iteration onwards, the resulting error will be really small $\sim0.02$. Hence, the room for improvement will be limited. The reader can find significant error reduction for other datasets in the appendix as well.

D. ASPECT execution time

So far, we have verified that ASPECT is effective in tweaking the features. Next, we will show that ASPECT is efficient as well. Fig.14 presents the running time of each tweaking permutation. Similar to the previous plots, the x-axis represents the dataset snapshot; y-axis represents the running time (minutes).

In Fig.14, the execution time increases linearly with the dataset size for most of the experiments. All tweaking permutations finish within 100 minutes. Moreover, different size-scalers result in different execution time. This is expected, as different size-scalers have different feature errors. Hence, the amount of tweaking is different.

For the same size-scalers and the same dataset, different tweaking permutations have different execution times. In general, L-C-P and L-P-C are more efficient than other tweaking permutations.

VI. LIMITATIONS AND OBSERVATIONS

ASPECT aims to tweak the features $F_1, F_2, \ldots, F_n$ so that the tweaked dataset has the corresponding target features $\tilde{F}_1, \tilde{F}_2, \ldots, \tilde{F}_n$. Extensive experiments above show that ASPECT has the capability of tweaking complex features with reasonably small errors within reasonable running time.

A. Limitations

While tweaking a feature, we might modify some already tweaked features. These already tweaked features may take various forms, which significantly increases the difficulty of proving some error bound of previously tweaked features. We state this issue as the Feature Tweaking Bound Problem:

Assuming a dataset $D_n$ has features $F_1, \ldots, F_n$. If a tweaking tool $T_{n+1}$ is applied on $D_n$, how much does it affect the previous features $F_1, \ldots, F_n$?

Solving the Feature Tweaking Bound Problem for general features might not be possible. Proving error bounds should be easier if the features satisfy certain properties. Consider the following trivial example: if each $F_i$ represents the attribute distribution of a distinct column, then one can easily see that tweaking tool $T_{n+1}$ will never affect $F_1, \ldots, F_n$. An example of a non-trivial restriction would be limiting the features to just 1 join.

Besides the Feature Tweaking Bound Problem, there is also the Feature Tweaking Order Problem:

When tweaking a dataset for $n$ features, which tweaking order results in the least error?

We believe the Feature Tweaking Bound Problem and Feature Tweaking Order Problem are issues that offer a rewarding challenge for research on dataset tweaking.

B. Observations

In developing ASPECT, we arrive at the following observations:

(O1) Non-overlapping features. If the features are not overlapping, then regardless of the modification of $T_{n+1}$, all the previous modified features will be preserved.

(O2) Determination of non-overlapping features. Given (O1), we would want to determine which features among
\( \tilde{F}_1, \ldots, \tilde{F}_n \) do not overlap. Then, the user will clearly know which features do not affect each other. This can be achieved in ASPECT through monitoring the dataset access by each tweaker. In ASPECT, each tweaker can only access the dataset via the functions similar to the ones provided in Fig.3. Hence, ASPECT knows if any two tweaking tools have accessed the same tuples. Then the problem is reduced to finding independent sets in graph theory, where the nodes are the tweaking tools. If two tweaking tools access the same tuples, then there will be an edge linking the two nodes. Even though, it is NP-Hard to find a maximum independent set, Robson [28] has proven that it can done in \( O(1.22^n) \) time, which is a reasonable complexity for a small number of tools \( n \).

(O3) Conflicting overlapping features. Overlapping features are called conflicting if no dataset can satisfy all of them. A simple example of conflicting features of a social network dataset is: \( F_1 = \) more than half of the customers are men and \( F_2 = \) more than half of the customers are women. Such features must be modified to resolve the conflict, and ASPECT always modifies the features that are applied earlier.

(O4) Non-conflicting overlapping features. For non-conflicting features, it is not always feasible to synthesize a dataset that satisfies all of them, even if one exists. For example, it is already NP-Hard to decide whether there exists a graph that satisfies certain degree distributions[5], so there is no polynomial algorithm that generates a graph for such distributions. For the sake of efficiency, we may have to sacrifice some feature accuracy. Even so, for the features in this paper, ASPECT maintains the features accurately. As we can see from Sec.V-C, the error is reduced to 0.02 after 2 to 3 iterations in the experiments.

VII. RELATED WORK

The Dataset Scaling Problem (DSP) was first advocated by Tay [32]. There have been several solutions to this problem in the field of relational database. UpSizeR [33] is the first solution to DSP, which uses attribute correlation extracted from an empirical dataset to generate a synthetic dataset. ReX [8] is a later work that scales up the original dataset by an integer factor \( s \), using an automated representative extrapolation technique. Chronos [17] scales the streaming data by focusing on capturing and simulating streaming data with both column correlation and temporal correlation. Recently, DSP was extended to non-uniform DSP (nuDSP) [36]. As a solution to nuDSP, DSCALER uses a correlation database which captures fine-grained, per-tuple correlations for scaling.

Data scaling is extended into other fields as well. In [25], the authors propose a dataset scaling problem for RDF data and provide a solution RBench that scales the original input dataset by preserving 4 features: resource identity (resource name, resource type, resource degree), relationship patterns (subgraphs with only relationship edges), predicate dictionary (frequency counts of the words) and attribute stars (frequency counts of the star structure). In [21], the authors lift the scaling approach from the pure database level to the OBDA level, where the domain information of ontologies and mappings are also taken into account as well. VIG [21] maintains the similarity for OBDA data by preserving the following features: size of columns clusters and disjointness, schema dependencies and column-based duplicates and NULL Ratios. However, VIG only supports dataset where each table has at most one foreign key only. In most storage systems, compression time and compression ratio are important issues. Hence, these two criteria should also be used for similarity measurement. In [15], SDGen is proposed to scale an input dataset to an arbitrary size which preserves these two similarities. In [37], the authors extend DSP to the Graph Scaling Problem for directed graphs. GSCALER is proposed as a solution that maintains not only local graph properties, e.g. degree distribution, but also global graph properties, e.g. effective diameter.

In the broader field, much work [7], [16], [26], [30] have been done on query-independent application-specific database generation (not scaling). The query-independent database generation refers to generating the database in terms of a given real data set (data-driven) or a set of its character descriptions (character-driven), which is initiated by Jim Gray [16].

For character-driven data generation, a parallel algorithm is first proposed to generate a dataset with a predefined schema and a predefined distribution [16]. However, no correlations between attributes are preserved. Later, Bruno and Chaudhuri introduce DGL [7], a simple specification language to generate datasets with complex synthetic distributions and inter-table correlations. In [20], Houkjaer proposes a table-wise graph model which holds various statistical information about the foreign key and column content. By using such a graph model, a more realistic dataset can be generated. PSDG [19] is another parallel solution to generate “industrial sized” dataset. PSDG supports easy parallelism, using a construct for specifying foreign keys. However, PSDG does not allow independent generation of dependent tables. Referenced tables have to be created for generating dependent tables. This is very inefficient if the referenced tables are not needed, and are huge in size. To overcome this, PDGF [27] is proposed.

Character-driven data generation also appears in graphs as well [2], [12], [22], [5]. For example, the Erdős-Rényi model generates a graph of any size \( n \) with a specified edge probability \( p \). gMark [5] is another tool that generates a graph database based on users’ specifications, e.g. degree distribution and node type occurrence. Recently, TrillionG [23] is proposed to generate large scale graphs in a short time. It can generate trillion-node graphs within two hours by using 10PCs.

For data-driven synthetic database generation, the input is always a real dataset. MDD [30] is the first tool which uses a real dataset for database generation. However, MDD uses very little information (name and address) from the real dataset. Similarly, TEXTURE [11] is another micro-benchmark for text query workloads. However, it only extracts simple properties, e.g. word distribution, document length. As an extension to PDGF, DBSynth [3] scales the dataset by utilizing the meta-data (e.g. min/max constraints) and the statistics from the input dataset. Strictly speaking, DSP is a sub-problem of query-independent synthetic database
There are also domain-specific benchmarks that generate datasets for specific domains. They include TPC [3], YCSB [9], LinkBench [4], LDBC [13], BigDataBench [34] and MWGen [35]. For example, MWGen [35] uses road and floor plans as input and generates a set of real world infrastructure together with moving objects in different transportation modes. Such domain-specific data generation usually generates only one fixed dataset with fixed schema for all applications under the same domain. Hence, domain-specific data generation is different from application-specific data scaling.

Nevertheless, previous works generate datasets with predefined features, instead of flexible features. To the best of our knowledge, ASPECT is the first framework which allows dataset scaling with flexible features through coordinating the tweaking tools.

VIII. CONCLUSION AND FUTURE WORK

This paper introduces ASPECT, a framework for flexible application of tweaking tools to enforce target features in synthetic dataset. To generate a scaled dataset with greater similarity comparing to the original dataset, one just needs to apply more tweaking tools. We demonstrate ASPECT by coordinating 3 highly overlapping and complex tweaking tools on real datasets to realize the target features. Extensive experiments show that ASPECT effectively reduces the errors by orders of magnitudes in the synthetic data without sacrificing efficiency.

ASPECT is a step towards the vision for application-specific benchmark data generation. It facilitates bottom-up collaboration among developers in contributing tools for tweaking synthetic datasets to enforce similarity with empirical data. Our current work is on the Feature Tweaking Bound and Order Problems. We hope to make some progress by restricting the feature types (e.g. single joins).

REFERENCES


Algorithm 3: Pairwise Feature Tweaking

1. for each $\rho^*_R$ do
2.   for each $(x, y) \in \Theta^-$ do
3.     while $\rho^*_R(x, y) \neq 0$ do
4.       $(x', y') \leftarrow \text{closest}(\Theta^+, (x, y))$
5.       $u, v \leftarrow \text{tupleRetrieve}(R, (x', y'))$
6.       $\text{tupleModification}(R, x' - x, y' - y, u, v)$
7.       $\text{statsUpdate}(\rho^*_R(x', y'))$
8.   end while
9.   $\text{statsUpdate}(\rho^*_R(x', y'))$
10. end for
11. end for

B. Theorems and proofs for linear feature

In this section, we present the formal proofs that are related to the linear feature. Define $H^* = H - \hat{H}$ and $h^*_{j,i} = h_{j,i} - \tilde{h}_{j,i}$ for all $1 \leq i, j \leq k$.

Lemma 1 (leadingElementAdjust): If the first $n$ rows of $H$ and $\hat{H}$ are the same, then $(h^*_{n+1,1}, h^*_{n+1,2}, \ldots, h^*_{n+1,n})$ can be tweaked to $(0, h^*_{n+1,2}, \ldots, h^*_{n+1,n})$, where $h^*_{n+1,1} \geq 0$ for $2 \leq i \leq n$.

Proof: There are two cases: $h^*_{n+1,1} > 0$ and $h^*_{n+1,1} < 0$.

Case $h^*_{n+1,1} > 0$: There are $h^*_{n+1,1}$ more tuples in $T_1$ having descendants in $T_{n+1}$. Hence, tweaking is needed to make these $h^*_{n+1,1}$ tuples have no descendants in $T_{n+1}$. It takes two steps: Leaf Tuple Plucking and Leaf Tuple Attaching.

Leaf Tuple Plucking: Consider $S_{n+1,1}$, the tuples in $T_1$ which have descendants in $T_{n+1}$. Let $R_{n+1,1}$ be the $h^*_{n+1,1}$ tuples from $S_{n+1,1}$ with least number of descendants in $T_{n+1}$.

Let $Q_{n+1,1}$ be the tuples in $T_{n+1}$ which are descendants of tuples in $R_{n+1,1}$.

Pluck all tuples in $Q_{n+1,1}$ by removing their foreign key reference to $T_n$. Then, all tuples in $R_{n+1,1}$ have no descendants in $T_{n+1}$, so $h^*_{n+1,1} = 0$ after the tuple plucking. Next, we will do Leaf Tuple Attaching.

Leaf Tuple Attaching: Let $V_{n+1,1}$ be the tuples in $T_n$ which are descendants of $S_{n+1,1} - R_{n+1,1}$. All tuples in $S_{n+1,1} - R_{n+1,1}$ already have descendants in $T_{n+1}$. Hence, attaching more tuples to $V_{n+1,1}$ will not change $h^*_{n+1,1}$. Therefore, attach back all the tuples in $Q_{n+1,1}$ by setting their foreign key reference randomly to tuples in $V_{n+1,1}$.

Case $h^*_{n+1,1} < 0$: There are $h^*_{n+1,1}$ more tuples in $T_1$ which do not have descendants in $T_{n+1}$. The tweaking takes two steps by doing tuple plucking first, and then tuple attaching.

Leaf Tuple Plucking: $|h^*_{n+1,1}|$ tuples in $T_{n+1}$ need to be found first. Once these $|h^*_{n+1,1}|$ tuples are plucked, we can attach them to the tuples in $T_{n+1}$ to increase $h^*_{n+1,1}$. For each tuple in $S_{n+1,1}$ (the tuples in $T_1$ which have descendants in $T_{n+1}$), pick 1 descendant in $T_{n+1}$ to form a leaf set $Leaf_{f_{n+1}1}$. It is obvious that plucking any tuples from $T_{n+1} - Leaf_{f_{n+1}1}$ never modify $h^*_{n+1,1}$. Then,

\[
|T_{n+1} - Leaf_{f_{n+1}1}| - |h^*_{n+1,1}|
\]

\[
= |T_{n+1}| - h^*_{n+1,1} - (h^*_{n+1,1} - h_{n+1,1})
\]

\[
= |T_{n+1}| - h_{n+1,1} \geq 0 \quad \text{(By L1)}
\]

Hence, $|h^*_{n+1,1}|$ tuples in $T_{n+1} - Leaf_{f_{n+1}1}$ can be randomly plucked. Next, we will attach these tuples back.

Leaf Tuple Attaching: Consider $S_{n,1} - S_{n+1,1}$, the set of tuples in $T_1$ having descendants in $T_n$ but no descendants in $T_{n+1}$. Since

\[
|S_{n,1} - S_{n+1,1}| - |h^*_{n+1,1}|
\]

\[
= |h_{n,1} - h^*_{n+1,1}| - (h^*_{n+1,1} - h_{n+1,1})
\]

\[
= h_{n,1} - h_{n+1,1} - (h^*_{n+1,1} - h_{n+1,1})
\]

\[
= h_{n,1} - h_{n+1,1}
\]

\[
= h_{n,1} - h_{n+1,1} \geq 0 \quad \text{(By L2)}
\]

Then, there are $|h^*_{n+1,1}|$ tuples from $S_{n,1} - S_{n+1,1}$, denoted as $Sub_{n+1}1$. For each tuple in $Sub_{n+1}1$, it must have a descendant in $T_n$. Hence, randomly attach the $|h^*_{n+1,1}|$ tuples from Leaf Tuple Plucking to the decent in $T_n$. Then all tuples in $Sub_{n+1,1}$ have descendants in $T_{n+1}$ now. Hence, $h^*_{n,1,i} = 0$ after attachment.

We are done with the leading element tweaking; next, we will prove the correctness non-leading element tweaking. Some care is needed to tweak $h^*_{n+1,2}$.

Suppose $h^*_{n,1} = 1$ for Fig.15, so we want to remove descendants in $T_D$ for 1 tuple in $T_B$. If we do this by plucking $d_5$ from $c_5$, then $h^*_{4,2}$ is decreased by 1 as well, so we should instead pluck $d_6$ or $d_7$. Therefore, when tweaking $h^*_{n+1,1}$, we should avoid affecting $h^*_{n+1,2}$.

If $h^*_{4,2} = -1$, we need to add descendants in $T_D$ for 1 more tuple in $T_B$. We can pluck a leaf, say $d_1$, and attach
\textbf{Leaf Tuple Attaching:} Similar to Lemma 1.

\textbf{Case }$h^*_{n+1,i} < 0$: Similar to Lemma 1, there are two steps: Leaf Tuple Plucking and Leaf Tuple Attaching.

\textbf{Leaf Tuple Plucking:} Similar to Lemma 1.

\textbf{Leaf Tuple Attaching:} If no isomorphic adjustment is needed, then it is the same as Lemma 1. Otherwise, based on Lemma 2, the maximum isomorphic adjustment is

$$|S_{n,i}| - |S_{n+1,i}| - (|S_{n,i-1}| - |S_{n+1,i-1}|)$$

Moreover,

$$|S_{n,i}| - |S_{n+1,i}| - (|S_{n,i-1}| - |S_{n+1,i-1}|) - |h^*_{n+1,i}| = h_{n,i} - h_{n+1,i} + h_{n+1,i-1} - h_{n,i-1} - \tilde{h}_{n+1,i} + h_{n+1,i} = h_{n,i} + h_{n+1,i-1} - h_{n,i-1} - \tilde{h}_{n+1,i} = \tilde{h}_{n,i} + \tilde{h}_{n+1,i-1} - \tilde{h}_{n,i-1} - \tilde{h}_{n+1,i} \geq 0 \quad \text{(By LA)}$$

Therefore, at least $|h^*_{n+1,i}|$ isomorphic adjustments in $T_i$ can be made. Let $Sub_{n+1,i}$ be the set of tuples that undergo isomorphic adjustments. For each tuple in the subset $Sub_{n+1,i}$, pick 1 descendant in $T_n$ and randomly attach a tuple plucked from the previous step. Hence, the leaf tuple attaching can be done.

\textbf{Theorem 6:} For a reference chain $T_k \rightarrow \cdots \rightarrow T_i$ in some $D_t$, let $H$ be the linear join matrix before tweaking and $\tilde{H}$ the target linear join matrix. If $\tilde{H}$ satisfies the necessary conditions in Theorem 1, then Algorithm 1 tweaks $H$ to give $\tilde{H}$.

\textbf{Proof:} Algorithm 1 iterates over the rows of $H^* = H - \tilde{H}$; for each row, the first entry is tweaked to 0 with Lemma 1, and the following entries are tweaked to 0 with Lemma 3.

\section{C. Theorem and proofs for coappear feature}

In this section, we present formal proofs that are related to coappear feature.

\textbf{Theorem 3 [sufficiency]} Suppose tables $T_1,\ldots, T_k$ reference the same tables $T^*_1,\ldots, T^*_n$. Let $\xi$ be the coappear distribution in some $D_t$ before tweaking and $\xi'$ the target coappear distribution. If $\xi'$ satisfies the necessary conditions in Theorem 2, then $T_{\text{coappear}}$ tweaks $\xi$ to become $\xi'$.

\textbf{Proof:} Both $\xi$ and $\xi'$ satisfy C1, so $|T_i|$ is unaffected by the tweaking. Similarly, C2 ensures $\sum_\nu \xi = \sum_\nu \xi'$, so $\sum_\nu \xi' = 0$. Therefore

$$\sum_{v \in \Delta^+} \xi'(v) + \sum_{v \in \Delta^0} \xi'(v) + \sum_{v \in \Delta^-} \xi'(v) = 0,$$

so $\sum_{v \in \Delta^+} \xi'(v) = \sum_{v \in \Delta^-} -\xi'(v)$, i.e. $\sum_{v \in \Delta^+} \xi'(v) = -\sum_{v \in \Delta^-} \xi'(v)$.

Each tweak decreases $\xi'(v')$ by 1 and increases $\xi'(v)$ by 1 for some $v' \in \Delta^+$ and $v \in \Delta^-$. After $\sum_{v' \in \Delta^+} \xi'(v')$ iterations, we get $\sum_{v \in \Delta^-} \xi'(v) = 0 = \sum_{v \in \Delta^-} -\xi'(v)$, so $\xi' = 0$. Hence, $\xi = \xi'$. 

\begin{figure}[h]
  \centering
  \begin{tabular}{cccc}
    \textbf{a} & \textbf{b} & \textbf{c} & \textbf{d} \\
    \textbf{1} & \textbf{2} & \textbf{3} & \textbf{4} \\
    \textbf{5} & \textbf{6} & \textbf{7} & \textbf{8} & \textbf{9} \\
    \textbf{1} & \textbf{2} & \textbf{3} & \textbf{4} \\
    \textbf{5} & \textbf{6} & \textbf{7} & \textbf{8} & \textbf{9} \\
  \end{tabular}
  \caption{Non-Leading Element Tweaking Demonstration}
\end{figure}
D. Theorem and proofs for pairwise feature

In this section, we present formal proofs that are related to linear feature. We first prove Theorem 5.

**Theorem 5** [sufficiency] For each response2post table $R$, let $\rho_R$ be the pairwise distribution in $D$ and $\tilde{\rho}_R$ the target pairwise distribution. If $\rho_R$ satisfies (P3) in Theorem 4, then Algorithm 3 tweaks $\rho_R$ to become $\tilde{\rho}_R$. Moreover, the extra tuples added to the $\text{post}$ table $P$ is at most $|U| - |P|$, where $U$ is the user table.

**Proof:** Since $\rho_R$ and $\tilde{\rho}_R$ both satisfy (P3), and $\mathcal{T}_{\text{pairwise}}$ does not affect $|U|$, we have $\sum_{x,y} \rho_R(x,y) = \sum_{x,y} \tilde{\rho}_R(x,y)$, and so $\sum_{x,y} \rho_R^*(x,y) = 0$. As in the proof of Theorem 3, this implies

$$\sum_{(x,y) \in \Theta^+} \rho_R^+(x,y) = \sum_{(x,y) \in \Theta^-} |\rho_R^+(x,y)|.$$

Each tweak by Algorithm 3 increases $\rho_R^+(x,y)$ and $\rho_R^+(y,x)$ by 1, and decreases $\rho_R^-(x',y')$ and $\rho_R^-(y',x')$ by 1 for $(x,y) \in \Theta^-$ and $(x',y') \in \Theta^-$. After $\frac{1}{2} \sum_{(x,y) \in \Theta^+} \rho_R^+(x,y)$ loops, we get $\sum_{(x,y) \in \Theta^+} \rho_R^+(x,y) = 0 = \sum_{(x,y) \in \Theta^-} |\rho_R^+(x,y)|$, so $\rho_R = 0$, i.e. $\rho = \tilde{\rho}$.

Moreover, if a new post needs to be added to $P$, then each user has at most 1 post. Thus, there are $|U| - |P|$ users who do not have posts, so that many posts need to be added to ensure each user has 1 post.

Previously, we assume a user does not respond to his/her own post. Now, we remove the assumption. However, we separate the distribution $\rho_R$ into 2 distributions: $\rho_S$ and $\rho_N$, where $\rho_S$ is the distribution generated by self-responding behavior, and $\rho_N$ does not contain any pairwise vector generated by self-responding. Sec.III-C has discussed the case for $\rho_N$, so we now discuss tweaking for $\rho_S$.

**Theorem 7:** For a response2post table $R$ in some $D_i$, let $\rho_S$ be the pairwise distribution generated by user self responding before tweaking, and $\tilde{\rho}_S$ the target pairwise distribution. If $\rho_S$ can be tweaked to become $\tilde{\rho}_S$, then $\tilde{\rho}_S$ satisfies the following conditions:

$$\begin{align*}
\text{(SP1)} & \quad \sum_{x} 2x\tilde{\rho}_S(x,x) + \sum_{x,y} (x+y)\tilde{\rho}_N(x,y) = 2|T_R| \\
\text{(SP2)} & \quad \sum_{x} \tilde{\rho}_S(x,x) = |U| \quad \text{where U is the user table.}
\end{align*}$$

**Proof:** (SP1) For each response tuple $t$, made from $u_i$ to $v_i$. If $u_i \neq v_i$, assuming their pairwise vector is $(x,y)$, then it is double-counted by $\tilde{\rho}_R(x,y)$ and $\tilde{\rho}_R(y,x)$. If $u_i = v_i$, then there are only $x$ tuples, which are double-counted by $(x+y)$. So we get the equality in (SP1).

(SP2) There are $|U|$ users, each user can respond to himself. Hence counted only once.

For tweaking $\rho_S$, it is similar to $\rho_N$.

Let $\rho_S^* = \rho_S - \tilde{\rho}_S$, $\Theta_S^* = \{(x,x) | \rho_S^*(x,x) > 0\}$ and $\Theta_S^- = \{(x,x) | \rho_S^*(x,x) < 0\}$. For each $(x,x) \in \Theta_S^-$, it adds $|\rho_S^*(x,x)|$ pairs $(u_i,v_i)$, where user $u_i$ has $x$ response2post tuples in $R$ referencing $u_i$’s post. It does this by looping $|\rho_S^*(x,x)|$ times, and in each iteration:

- **PairwiseVectorRetrieve:** Pick $v' = (x',x') \in \Theta_S^+$ that is closest to $(x,x)$ by Manhattan distance.
- **TupleModification:** Choose users $u$ with self-respond pairwise vector $(x',x')$ and tweak $u$’s responses to $u$’s post, as follows: If $x < x'$: this means $u$ has $x' - x$ more responses to $u$’s post than desired, so $\mathcal{T}_{\text{pairwise}}$ randomly chooses and removes $x' - x$ such responses. If $x > x'$: $\mathcal{T}_{\text{pairwise}}$ adds $x' - x$ responses from $u$ on $u$’s post. If $u$ has no post, we artificially create a post for $u$. To do this, we pick another user $w$ who has more than 1 post and, among $w$’s posts, pick a post $p_w$ with minimum responses; we make $p_w$ a post by $u$, and shift the responses to $p_w$ to other posts by $u$. If (in the worst case) all other users have at most 1 post, then we create a new post $p$ for $u$, and add $x' - x$ responses to $p$.
- **StatsUpdate:** Increase $\rho_S^*(x,x)$ by 1 and decrease $\rho_S^*(x',x')$ by 1.

The following theorem says that conditions in Theorem 7 suffices to ensure that $\mathcal{T}_{\text{pairwise}}$ tweaks $\rho_S$ to become $\tilde{\rho}_S$.

**Theorem 8:** For each response2post table $R$ in some $D_i$, let $\rho_S$ be the self-responded pairwise distribution before tweaking and $\tilde{\rho}_S$ the target pairwise distribution. If $\rho_S$ satisfies the conditions in Theorem 7, then $\rho_S$ can be tweaked to $\tilde{\rho}_S$. Moreover, the extra tuples added to the $\text{post}$ table $P$ is at most $|U| - |P|$, where $U$ is the user table.

**Proof:** The proof is similar to the proof in Sec.III-C.

E. Dataset summary

In this section, we summarize the datasets used in the experiments. We used 4 datasets from Douban4 and Xiami5. Douban is a Chinese social network website that allows the creation and sharing of content related to movies, books, music, recent events and activities in Chinese cities. Xiami is a Chinese online music website that provides recommendations of music services, offline music activities, and other interactive content. The short summary of the 4 datasets are the following:

1. **DoubanMovie** contains movie-related data in 17 tables, with table sizes ranging from 10856 to 36747342 tuples.
2. **DoubanBook** contains book-related data in 12 tables, with table sizes ranging from 686605 to 12891598 tuples.
3. **DoubanMusic** contains music-related data in 10 tables, with table sizes ranging from 52078 to 7086936 tuples.
4. **Xiami** also contains music-related data, but is larger: It has 26 tables and more than 90millions tuples.

Fig.16 presents the dataset size for each partition. For example, the 6th partition of DoubanMovie $D_6$ is 2.5 Gigabytes.

The schema of each dataset is presented as follows.

4https://www.douban.com
5https://www.xiami.com
Feature similarity for DoubanMovie, DoubanMusic, DoubanBook

In this section, we present the feature similarity for DoubanMovie, DoubanMusic, and DoubanBook.

1) Linear feature similarity: Fig. 21 presents the linear feature similarity results. All tables are involved in at least
Fig. 21: Linear feature errors

Fig. 22: Coappear feature errors

one linear join matrix for all datasets. For DoubanMovie, each of the 17 tables is involved in one of 24 linear join matrices. For example, Movie_Comment → Movie and Trailer_Comment → Trailer → Movie are maximal linear joins. Similarly, the 12 tables in DoubanBook have 15 linear join matrices, the 11 tables in DoubanMusic have 14 linear join matrices.

In general, the later $T_{\text{linear}}$ is applied, the smaller the linear feature error, i.e. C-L-P and P-L-C have smaller errors than L-C-P and L-P-C, and C-P-L and P-C-L have 0 error. All permutations reduce the error tremendously for all size-scalers on all datasets.

Even through the error reduction is huge, there are still some cases that the error is $>0.1$. For example, Rand-DoubanBook for L-P-C. It reduces linear feature error from 2 to 0.2. We further investigate this issue, the largest error occurs on the join Book_Comment → User. For L-P-C, while tweaking the coappear distribution for $\xi_T$, where $T$ is (Book_Comment, Book_Read, Book_Reading, Book_Wish, Book_Review), it overlaps with 1 pairwise distribution — Book_Review as a post table, and Review_Comment as a response2post table. Moreover, it overlaps with 12 linear joins (e.g. Book_Comment → User, Book_Comment → Book). As stated in Sec.V-A, such highly overlapped features increase the difficulty of getting a validated modification as described in Section II. Hence, this could be a potential reason that error is $>0.1$.

2) Coappear feature similarity: Fig. 22 presents the coappear feature similarity results. There are 6 coappear distributions, DoubanBook has 4 and Xiami has 12. In each case, each table is involved in one or more coappear distributions.

Fig. 22 shows that, like for $T_{\text{linear}}$, the later $T_{\text{coappear}}$ is applied in the tweaking order, the smaller the coappear error. In general, we find that permutations where $T_{\text{coappear}}$ is after $T_{\text{linear}}$ reduces the errors more than if $T_{\text{coappear}}$ is before $T_{\text{linear}}$. This is expected, since $T_{\text{linear}}$ modifies the coappearing tables massively after $T_{\text{coappear}}$ is done.

For average error, all permutations of tweaking significantly reduce the error for all datasets for all size-scalers. It is below 0.1 for all most tweaking.

For the plot Dscaler-DoubanMovie, we observe that the tweaking permutations ($T_{\text{linear}}$ applied after $T_{\text{coappear}}$) have an error around 0.2. By looking at the details, we find that this happens for the coappear distribution involving many tables. Take $\xi_T$ for example, where $T$ is (Movie_Comment, Movie_Seen, Movie_Watching, Movie_Wish, Movie_Review, Movie_Photo). This coappear distribution overlaps with 12 linear join matrices and 2 pairwise distribution. This coappear distribution will be modified by 12 linear tweaking tools if $T_{\text{linear}}$ applied after $T_{\text{coappear}}$. Hence, increase the difficulty of getting a validated modification as described in Section II. Nevertheless, we still have a small error for ReX-DoubanMovie, Rand-DoubanMovie for such a highly overlapped structure.

For ReX-DoubanMovie, even through the error without tweaking is as low as 0.01. All tweaking permutations are still able to reduce the error.

3) Pairwise feature similarity: Fig. 23 presents the pairwise feature similarity results. DoubanMovie has 2 pairwise distributions: (i) Review as post table and Review_Comment as response2post table; and (ii)
**G. Query similarity for DoubanMovie, DoubanMusic, DoubanBook**

In this section, we similarly run queries on DoubanMovie, DoubanMusic, DoubanBook, and compare the query results on ground-truth dataset and scaled dataset.

1) **Query similarity for DoubanMovie**: Fig.24 presents the query results on DoubanMovie. The 4 queries used are: $Q_1$ computes the number of movies that have video clips with commenters; $Q_2$ computes the number of movies that have been commented on by at most 10 different users; $Q_3$ computes the average number of stars per movie; $Q_4$ computes the number of user pairs having interactions through a movie review.

As we can see from Fig.24, all tweaking permutations reduce the query error significantly on both size-scalers. The errors are reduced to $< 0.05$ for most of the tweaking permutations.

2) **Query similarity for DoubanMusic**: Fig.25 presents the query results on DoubanMusic. The 4 queries used are: $Q_1$ computes the number of users that have written a album-view with commenters; $Q_2$ computes the number of stars that have at most 10 different fans; $Q_3$ computes the average number of interested listeners of a album; $Q_4$ computes the number of user pairs having interactions through a album review.

Similar to DoubanMovie, all permutations reduce the errors tremendously.

3) **Query similarity for DoubanBook**: Fig.26 presents the query results on DoubanBook. The 4 queries used are: $Q_1$ computes the number of users that have written a book-view with commenters; $Q_2$ computes the number of diaries that have at most 10 different commenters; $Q_3$ computes the average number of interested readers of a book; $Q_4$ computes the number of user pairs having interactions through a book review.

As we can see from Fig.26, most of the tweaking permutations reduce the errors tremendously except for few rare cases, e.g. Dscaler-DoubanBook-$Q_1$. For the L-C-P permutation, we can see that it has a larger error than the baseline. This is expected, since $Q_1$ is a linear feature related query, and the linear feature that were tweaked by $T_{\text{linear}}$ is subsequently modified by $T_{\text{coapper}}$ and $T_{\text{pairwise}}$. Such a scenario can be improved by having more iterations. In Fig.27, we run L-C-P on Dscaler-DoubanBook with more iterations.
Fig. 26: Query similarity for DoubanBook

<table>
<thead>
<tr>
<th></th>
<th>No-Tweak</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>D6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First iteration</td>
<td>0.0787</td>
<td>0.1197</td>
<td>0.1876</td>
<td>0.2682</td>
<td>0.2914</td>
<td></td>
</tr>
<tr>
<td>Second iteration</td>
<td>0.0001</td>
<td>0.0007</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>Third iteration</td>
<td>0</td>
<td>0.0004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

We can see that from second iteration onwards, the Q1 error is reduced to less than 0.001.

Fig. 27: Query similarity improvement with more iterations. We run L-C-P for more iterations on Dscaler–DoubanBook, and test Q1.

We can see that, the execution time increases linearly with the dataset size for most of the experiments. DoubanMovie is the largest dataset, it takes more time. Nevertheless, most experiments finishes with 60 minutes for the largest snapshot of DoubanMovie. DoubanMusic and DoubanBook are the smaller datasets, hence, it takes less time, within 60 minutes, for the worst tweaking permutation.

For the same dataset, different size-scaler will result in different execution time. This is understandable, the data generated by the size-scalers have different feature errors. Hence, the amount of tweaking is different. For the execution time for each permutation varies among the different size-scaler. Moreover, for the same size-scaler and the same dataset, different tweaking permutation has different execution time. In general we find that L-C-P and L-P-C are more efficient than other tweaking permutations.

H. Similarity improvement over iterations

In this section, we present feature similarity results for different iterations. Fig.28, Fig.29 and Fig.30 present the results of running 6 tweaking permutations for up to 4 iterations on the dataset generated by DSCALER, ReX and Rand. Take Fig.28 for example, for coappear feature, 4th column for C-L-P is 0.031. It means that after running C-L-P permutation on the data generated by DSCALER for 4 times, the coappear feature error is 0.031. It is a 10-fold decrease from 0.306 (the No-Tweak baseline).

For all the three figures, we can see that the more iterations of tweaking, the less error we will have. On average, ASPECT can achieve an error of around 0.02 after 2 or 3 iterations.

I. Execution time for DoubanMovie, DoubanMusic, DoubanBook

We can see that, the execution time increases linearly with the dataset size for most of the experiments. DoubanMovie is the largest dataset, it takes more time. Nevertheless, most experiments finishes with 60 minutes for the largest snapshot of DoubanMovie. DoubanMusic and DoubanBook are the smaller datasets, hence, it takes less time, within 60 minutes, for the worst tweaking permutation.

For the same dataset, different size-scaler will result in different execution time. This is understandable, the data generated by the size-scalers have different feature errors. Hence, the amount of tweaking is different. Take DoubanMovie for example, the execution time for each permutation varies among the different size-scaler. Moreover, for the same size-scaler and the same dataset, different tweaking permutation has different execution time. In general we find that L-C-P and L-P-C are more efficient than other tweaking permutations.