A Collaborative Framework for Similarity Enforcement in Synthetic Scaling of Relational Datasets

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

# School of Computing, National University of Singapore

1dcszj@nus.edu.sg
2dcstayyc@nus.edu.sg

Abstract—Researchers and developers use benchmarks to compare their algorithms and products. A database benchmark must have a dataset. To be application-specific, this dataset should be empirical. However, previous work typically extracts a fixed set of properties \( \Pi = \{ \pi_1, \pi_2, \ldots, \pi_n \} \) from a top-down design of domain-specific benchmarks. A different approach \( (D \rightarrow \Pi \rightarrow \tilde{D}) \) becomes increasingly intractable as \( \Pi \) gets larger, so a new solution is necessary.

This paper proposes ASPECT, a framework for scaling datasets \( D \) to \( \tilde{D} \) and enforcing similarity between \( D \) and \( \tilde{D} \). Different from previous approaches, ASPECT is a repository for scaling and enforcement tools that can be contributed by users or developers in the database community. The framework provides a uniform API to facilitate collaboration and interoperability among these tools.

With ASPECT, \( D \) is first scaled up, or down, by a size-scaler \( S_0 \). The user then selects a set of target similarity properties \( \Pi = \{ \tilde{\pi}_1, \ldots, \tilde{\pi}_n \} \). For each desired property \( \tilde{\pi}_k \), there is a tool \( T_n \) that tweaks the synthetic data in \( D \) to ensure \( D \) has the desired \( \tilde{\pi}_k \). ASPECT coordinates how \( T_1, \ldots, T_n \) tweak \( D \), so \( T_n(\cdot)(\cdot)(\cdot)(\cdot) \) has the properties \( \tilde{\pi}_1, \ldots, \tilde{\pi}_n \).

By shifting from \( D \rightarrow \Pi \rightarrow \tilde{D} \rightarrow D \rightarrow \Pi \), data scaling becomes flexible: The user can customize the similarity between \( D \) and \( \tilde{D} \) by selecting the desired properties from the toolset. Extensive experiments on real datasets show that ASPECT can enforce similarity in the dataset effectively and efficiently.

I. INTRODUCTION

Benchmarks are ubiquitous in the computing industry and academia. Developers use them 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) 1. However, the small number of TPC benchmarks is increasingly irrelevant to the myriad of diverse applications, and the TPC standardization process is too slow [32]. 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 [33].

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.

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 [25], [34]. For all cases, \( \tilde{D} \) must be similar to \( D \).

To ensure \( \tilde{D} \) is similar to \( D \), previous work [18], [25], [34], [37] typically follow the framework in Fig. 1. Each algorithm extracts a fixed set of properties \( \Pi = \{ \pi_1, \pi_2, \ldots, \pi_n \} \) from \( D \), then scales \( \Pi \) to the desired property for the scaled dataset \( \tilde{D} \). \( \tilde{D} \) is finally synthesized based on \( \Pi \). Here defines the similarity between \( D \) and \( \tilde{D} \); the more properties in \( \Pi \), the greater the similarity between \( D \) and \( \tilde{D} \).

For example, if \( D \) is a graph and \( \Pi = \{ \pi_1, \pi_2 \} \), where \( \pi_1 \) is density and \( \pi_2 \) is number of triangles, then we would expect \( D \) and \( \tilde{D} \) to be similar in terms of density and triangles.

A. Limitations of current approaches

From the perspective of a developer or user, there are limitations to current approaches.

Code reusability: Consider the scenario where one application developer implements an algorithm \( A_{12} \) using the property set \( \{ \pi_1, \pi_2 \} \). Later, another developer may find it more important for her application to preserve \( \{ \pi_2, \pi_3 \} \), where \( \pi_3 \) is the number of rectangles. So she implements another algorithm \( A_{23} \) to preserve \( \{ \pi_2, \pi_3 \} \). However, a third developer might want to preserve \( \{ \pi_1, \pi_2, \pi_3 \} \); what should he do? In this case, \( A_{12} \) or \( A_{23} \) only preserves part of \( \{ \pi_1, \pi_2, \pi_3 \} \). To preserve \( \{ \pi_1, \pi_2, \pi_3 \} \), one has to modify \( A_{12} \) to preserve the extra \( \pi_3 \), or modify \( A_{23} \) to preserve the extra \( \pi_1 \), or write a

---

1http://www.tpc.org/
new algorithm $A_{123}$ from scratch. This is a waste of effort, since $\pi_1$, $\pi_2$, $\pi_3$ were already preserved by $A_{12}$ and $A_{23}$.

**Code scalability**: As mentioned previously, the more properties in $P$, the greater the similarity between $D$ and $\bar{D}$. However, a large property set dramatically increases the difficulty of designing an algorithm that maintains the properties simultaneously. For example, if $P = \{\pi_1, \pi_2, \pi_3, \pi_4\}$, where $\pi_4$ is the fraction of nodes with degree 1, then it is less likely that a developer can design a single algorithm which preserves all 4 properties. If we only consider degree distributions as properties, then it is already NP-hard to decide whether there exists a graph satisfying certain degree distributions [5].

**Property choice**: With current approaches, once an algorithm is implemented, the properties are fixed. In the example above, suppose software $A_{12}$ and $A_{23}$ exist to preserve $\{\pi_1, \pi_2\}$, and $\{\pi_2, \pi_3\}$, respectively. Given $A_{12}$ and $A_{23}$, a user cannot choose to preserve $\{\pi_1, \pi_2, \pi_3\}$, nor $\{\pi_1, \pi_3\}$.

**B. Overcoming the limitations**

This paper proposes ASPECT, a flexible framework that avoids hardwired property enforcement in synthetic dataset scaling. Unlike existing approaches, ASPECT takes the following two steps, as illustrated in Fig. 2:

**Step 1**: Use an off-the-shelf size-scaler $S_0$ to scale $D$ to $\bar{D}_0$ of desired size. $S_0$ may fail to preserve some desired properties.

**Step 2**: For the desired property set $\{\bar{\pi}_1, \bar{\pi}_2, \ldots, \bar{\pi}_k\}$, apply independently developed tools $T_1, T_2, \ldots, T_k$ on $\bar{D}_0$. Each tool $T_i$ generates a dataset $\bar{D}_i$ by adjusting $\bar{D}_{i-1}$. After the adjustment of $T_i$, $\bar{D}_i$ satisfies $\{\bar{\pi}_1, \bar{\pi}_2, \ldots, \bar{\pi}_i\}$. Note that Step 2 does not depend on the size-scaler in Step 1. We call this **tweaking** $\bar{D}_{i-1}$ by tool $T_i$. The final dataset $\bar{D}_k$ is $\bar{D}$.

ASPECT easily resolves the above-mentioned limitations: For **code reusability**, each property tweaking tool is independently developed. If some developer $X$ has implemented tools $T_1$ and $T_2$ to enforce properties $\bar{\pi}_1$ and $\bar{\pi}_2$, and someone else $Y$ wants to enforce properties $\bar{\pi}_2$ and $\bar{\pi}_3$, then $Y$ just needs to implement a tool $T_3$ for $\bar{\pi}_3$, and use $T_2(T_1(D))$ or $T_3(T_2(D))$. For **code scalability**, to preserve the property set with $n$ properties, the developer just needs to implement $n$ tweaking tools, instead of hard-coding all $n$ properties and integrating them into a single piece of software. For the issue of **property choice**, once tweaking tools $T_1$, $T_2$ and $T_3$ are available for $\bar{\pi}_1$, $\bar{\pi}_2$ and $\bar{\pi}_3$ respectively, the user just needs to apply tools accordingly. For example, to preserve $\{\bar{\pi}_1, \bar{\pi}_2, \bar{\pi}_3\}$, one can use $T_1(T_2(T_3(D)))$; to preserve $\{\bar{\pi}_1, \bar{\pi}_3\}$, one can use $T_1(T_3(D))$.

Hence, to enforce greater similarity in the scaled dataset $\bar{D}$, we just need to apply more tweaking tools. Since there are innumerable applications and properties, and the list is ever-growing, we envision having developers in the database community contribute tweaking tools to ASPECT. As the list of applications and properties grow, this repository will also collect more tools for users to customize their scaled datasets. This would go some way towards realizing the suggested paradigm shift to a bottom-up collaboration for application-specific benchmarking [33].

However, the tools require some coordination, since some changes to $\bar{D}_{i-1}$ by one tool $T_i$ may be undone by another tool $T_j$. Moreover, since the tools are implemented independently by different developers, using different tweaking techniques, ASPECT must specify the interfaces and provide a framework for the tools to collaborate and interoperate.

**C. Overview**

To summarize, we make four contributions:

1. We propose ASPECT, a framework for flexible enforcement of property similarity in synthetic datasets.
2. We present results from extensive experiments on real datasets, to verify that ASPECT can effectively and efficiently enforce similarity in a scaled dataset.
3. We present necessary and sufficient conditions, and tweaking algorithms, for three nontrivial properties.
4. We state a **Property Tweaking Bound Problem** and a **Property Tweaking Order Problem** that offer a rewarding challenge for research on dataset tweaking.

We begin by surveying related work in Sec. II, then introduce the ASPECT architecture in Sec. III. Sec. IV explains the challenges in analyzing the errors. We introduce three new complex properties that serve to illustrate the ASPECT framework in Sec. V. 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 [33].Sec. VI describes the datasets and similarity measures used in the experiments, and the results are presented in Sec. VII. Sec. VIII points out some limitations and observations of ASPECT. Sec. IX concludes with a summary.

**II. RELATED WORK**

The Dataset Scaling Problem (DSP) was first advocated by Tay [33]. Several solutions have been proposed in the case of relational databases.

The first solution is UpSizeR [34], 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 [18] scales 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) [37]; as a solution to nuDSP, Dscaler uses a correlation database that captures fine-grained, per-tuple correlations for scaling.

Dataset scaling is extended to other fields as well. In [26], the authors propose a scaling problem for RDF datasets and
provide a solution (RBench) that scales the 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 [22], 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. VIG [22] maintains the similarity for OBDA data by preserving the following features: size of column clusters and disjointness, schema dependencies and column-based duplicates and NULL Ratios. However, VIG only supports datasets where each table has at most one foreign key.

In most storage systems, compression time and compression ratio are important issues. so these two criteria may also be used for similarity enforcement. In [16], SDGen scales an input dataset to an arbitrary size while preserving similarity in compression time and ratio.

In [38], the authors extend DSP to graph datasets with a Graph Scaling Problem for directed graphs. G scaler is proposed as a solution that maintains not only local properties (e.g. degree distributions), but also global properties (e.g. effective diameter).

In the broader field, much work [7], [17], [27], [31] have been done on query-independent application-specific database generation (not scaling). This approach generates the database in terms of a given real dataset (data-driven), or a set of its characteristic descriptions (characteristic-driven), which is initiated by Jim Gray [17].

For characteristic-driven data generation, a parallel algorithm is first proposed to generate a dataset with a predefined schema and a predefined distribution [17]. 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 [21], Houkjaer et al. propose a table-wise graph model that 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 [20] is another parallel solution to generate “industrial sized” datasets. 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 [28] is proposed.

There is characteristic-driven data generation for graphs as well [2], [5], [13], [23]. For example, the Erdos-Renyi model generates a graph of any size \( n \) with a specified edge probability \( p \). gMark [5] is another tool that generates a graph based on user specifications, e.g. degree distribution and node type occurrence. Recently, TrillionG [24] is proposed to quickly generate large-scale graphs; e.g. it can generate a trillion-node graph in 2 hours by using 10 personal computers.

For data-driven synthetic database generation, the input is always a real dataset. MUDD [31] is the first tool which uses a real dataset for database generation. However, MUDD uses very little information (name and address) from the real dataset. Similarly, TEXTURE [12] is a 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 database 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 generation (data-driven).

There are also domain-specific benchmarks that generate datasets for specific domains. They include TPC, YCSB [10], LinkBench [4], LDBC [14], BigDataBench [35] and MWGen [36]. For example, MWGen [36] 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. Thus, domain-specific data generation is different from application-specific data scaling.

Most importantly, previous work generate datasets with predefined features, without offering a choice of features. To the best of our knowledge, ASPECT is the first framework that allows dataset scaling with a flexible choice of features, through the coordination of data tweaking tools.

### III. ASPECT Architecture

As shown in Fig. 2, ASPECT has two stages: size scaling and property enforcement.

#### A. Size scaling

ASPECT first runs a tool \( S_0 \) to scale the input empirical \( D \) to a synthetic \( D_0 \) of the desired size. Note that \( S_0 \) could be any tool that guarantees the number of tuples in each table generated is as expected, and there are no invalid foreign key values. For example, \( S_0 \) may be Dscaler [37], or ReX [8].

#### B. Property enforcement

After the dataset is resized, ASPECT then coordinates the application of tools \( T_i \) on \( D_{i-1} \), to make sure the property \( \pi_i \) is reflected in the tweaked dataset \( D_i = T_i(D_{i-1}) \). In the tweaking process, there are a few issues:

1. How do we get the target property \( \pi_i \)?

---

**Fig. 3: Frequently Used Notation**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D/D )</td>
<td>original/final database</td>
</tr>
<tr>
<td>( \Pi )</td>
<td>property set extracted from ( D )</td>
</tr>
<tr>
<td>( \Pi )</td>
<td>target property set expected to be present in ( D )</td>
</tr>
<tr>
<td>( \pi_i )</td>
<td>one individual property extracted from ( D )</td>
</tr>
<tr>
<td>( \pi_i )</td>
<td>one target property expected to be present in ( D )</td>
</tr>
<tr>
<td>( T_i )</td>
<td>the tweaking tool that tweaks the scaled database to enforce ( \pi_i ) after the tweaking</td>
</tr>
</tbody>
</table>
We now explain how each individual tool $T_i$ should be implemented. Each $T_i$ must have at least 5 components.

**Target Generator:** This module generates the target property statistics for the tweaked dataset. Such a generation can be done in 3 ways:

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

(b) **Developer Generation** A developer who implements $T_k$ may have a better understanding of how $\pi_k$ (e.g. the number of comments per post) changes when the dataset scales up or down. The developer may, therefore, provide the code for generating $\pi_k$.

(c) **Statistical Extrapolation** Some datasets may have a time attribute that can be used to take snapshots $D_1, \ldots, D_r$ of $D$. Otherwise, the user can provide some sampling technique to obtain $D_1, \ldots, D_r$ (with increasing size), or accept the default VDFS [9] sampling provided by ASPECT. ASPECT then extracts statistics of the desired property from $D_1, \ldots, D_r$, fits these statistics with some distribution, and extrapolates the distribution parameters to get the target property. For example, the number of comments per post in $D_1$ may be modeled by a Poisson($\lambda_1$) distribution, and $\lambda_1, \ldots, \lambda_r$ fitted by some polynomial that is used to determine the target $\lambda$ for $\bar{D}$. 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. $\langle \text{weight, height} \rangle$ or (age, income, gender)).

**Tweaking Algorithm:** It tweaks $\bar{D}_{L-1}$ to make sure that the target property $\pi_i$ is enforced in $\bar{D}_i$ when the algorithm terminates. Note that some properties (e.g. the fraction of user-pairs who comment on each other’s posts) require nontrivial tweaking algorithms. Tweaking algorithm proposes the modifications for Property Validator and Statistics Updater, and the execution of the modifications is done through Statistics Updater, which is illustrated in Fig. 5.

**Property Evaluator:** It calculates the property statistics for $\pi_i$ from a given dataset $\bar{D}_i$.

**Property Validator:** It checks whether a proposed tuple insertion/deletion/replacement adversely affects an existing property. Suppose, to enforce a property $\pi_k$, the tool $T_k$ proposes to modify a tuple $t$ to become $t'$, but $t'$ will cause a violation of some currently enforced $\pi_i$. Then the Property Validator for $T_i$ will vote against $t'$ and $T_k$ 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.

**Statistics Updater:** It first modifies the tuples as proposed by Tweaking Algorithm, then updates the relevant statistics for each property.

It is the developer’s responsibility to ensure that a tool correctly implements the 5 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.
D. ASPECT API

As mentioned in §4, we need to guarantee each individually developed tool can be used in ASPECT. So, we standardize the API for Statistics Updater and Property Validator. By having the same structure, all tweaking tools will be compatible with ASPECT. Fig. 5 presents the common functions to be implemented for any Statistics Updater. Property 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. 6, 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 \( \langle v_1, \ldots, v_d \rangle \) (colValues), some columns \( \langle c_1, \ldots, c_d \rangle \), this operation adds this \( \langle v_1, \ldots, v_d \rangle \) 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 to 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. 6, 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. 6, Step3 is a replaceValues operation. It replaces the first, second and third columns of second and third tuples with \([7,7,7]\).

Any property tweaking tool developer has to implement the functions in Fig. 5 to update the corresponding statistics. These modifications in ASPECT are not exhaustive. There may be other types of modifications that one wants to make, but the developer has to transform other modifications to the three basic operations in Fig. 5. We thus sacrifice some accuracy to favor generality for ASPECT and ease of programming for the developers.

If a tool \( T_1 \) wants to validate a modification (replace \( a_1 \) with \( a'_1 \) and \( b_1 \) with \( b'_1 \)) on two tables A and B, \( T_1 \) can validate it on table A and then validate it on table B, then accepts the modifications if both tables’ validations are successful. However, successful validations on table A and table B individually do not imply that the modifications on table A and table B are valid together. We may have such false-positive/false-negative cases for these types of validations, but we believe the effect is minor.

We illustrate this in Sec. V-B, where we modify \( k \) tables simultaneously. To fit into ASPECT, we validate/modify the tables one by one. Experiments in Sec. VII show this approach is effective.

IV. CHALLENGES IN ERROR ANALYSIS

In ASPECT, the more tools we apply, the more properties we can preserve. As pointed out previously, for some property \( \pi_j \), it is inevitable that some previously tweaked property \( \pi_i \) may be affected when tweaking \( \pi_j \). The first concern is how much \( \pi_i \) is affected while tweaking \( \pi_j \) under ASPECT.

It is difficult to analyze the errors formally. The main challenges are, in the ASPECT approach

- the empirical dataset is unknown,
- the properties are arbitrary, and
- the tweaking algorithms are unforeseeable.

Firstly, for the same properties and same tweaking algorithms applied on different datasets, the smallest error can be different.

Take Fig. 7 for example, there are two properties \( \pi_1 \) and \( \pi_2 \) expected to be reflected in the final dataset. In this example, there are 2 different empirical datasets, each dataset has 2 columns, where the first column contains digits and the second column contains letters. It is easy to see that \( \pi_1 \) is already reflected in both datasets. Now we are going to apply \( T_2 \) to fix \( D_2 \), where \( T_2 \) randomly chooses a \( B \) and converts it to \( A \).

For the first dataset, as presented, \( T_2 \) first converts the third \( B \) to \( A \). Then, \( T_2 \) converts the second \( B \) to \( A \). At the end of tweaking, the dataset satisfies both \( \pi_1 \) and \( \pi_2 \). For the second dataset, regardless of the choices of \( B, \pi_1 \) has to be violated.

We thus see that different datasets can yield different errors for the same properties and tweaking algorithm.

Secondly, even if we apply the same tweaking algorithms on the same dataset for the same set of properties, the resulting error can still be different.

Fig. 8 uses the same properties and tweaking algorithms used in Fig. 7. It presents 2 different executions of the tweaking algorithm \( T_2 \) on the same dataset.
For the first execution, $\mathcal{T}_2$ first converts the third $B$ to $A$. Then, $\mathcal{T}_2$ converts the second $B$ to $A$. For the second execution, if $\mathcal{T}_2$ first converts the first $B$ to $A$. Then, we need to convert another $B$ to $A$. Regardless of the choice of such $B$, $\mathcal{T}_2$ has to violate $\pi_1$.

As we can see, for such a simple example, we can even have different errors for the same tweaking algorithm applied on the same dataset to preserve the same properties.

In Sec. VIII, we crystallize the difficulty in this error analysis by stating a Property Tweaking Bound Problem, and leave the solution to future work.

V. SIMILARITY ENFORCEMENT FOR COMPLEX PROPERTIES: THREE EXAMPLES

Instead of a formal error analysis, we now demonstrate empirically how $\pi_1$ is affected while tweaking $\pi_1$. We apply 3 tools $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$ sequentially on $\mathcal{D}_0$ to enforce properties $\pi_1, \pi_2, \pi_3$. At the end of the tweaking, we will examine how well the three properties are preserved. We use two criteria to select the 3 properties:

**Popularity**: Our ultimate goal is to build a framework for application-specific dataset scaling, so it only makes sense if the properties are popular. Indeed, the 3 chosen properties — *linear*, *coappear* and *pairwise* — are used widely in the literature [1], [15], [19], [30].

**Complexity**: We want to check how well ASPECT can maintain properties, given that one tool can undo the work of previously applied tools. Therefore, the properties we use for demonstration should be complex and affect each other.

Tools for simple properties, such as “number of null values in a column” or “number of tuples in each table”, are easy to implement; they are already in the current version of ASPECT, and users can apply them if needed.

To avoid presenting a strawman test, we skip such simple properties in this paper, as the corresponding tools do not interfere with each other. Instead, we choose *linear*, *coappear* and *pairwise*. Incidentally, to the best of our knowledge, we are the first to provide solutions to preserving these 3 properties.

For any property $\pi_i$, we look at 2 issues:

- Is it possible to tweak the current dataset to enforce $\pi_i$? (necessity)
- How to enforce $\pi_i$? (sufficiency)

A. Linear property

Applications are often interested in computing $T_k \bowtie \ldots \bowtie T_1$ for some reference chain $T_k \rightarrow \ldots \rightarrow T_1$. Variants of linear joins are widely used to generate query result approximations, such as database sampling [15] and database generation [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*. This is what we call a linear property.

Fig. 9 illustrates the concept of a linear property. For any reference chain $T_k \rightarrow \ldots \rightarrow T_1$, the linear property describes how one tuple $t_i \in T_1$ is transitivity referenced by other tuples $t_j \in T_j$, for any $j > i$. In Fig. 9, $a_2, a_3 \in T_A$ 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 $T_D$.

**Definition 1**: $T_k \rightarrow \ldots \rightarrow T_1$ is called a reference chain if $T_k$ references $T_{k-1}, T_{k-1}$ references $T_{k-2}, \ldots, T_2$ references $T_1$. $T_k \rightarrow \ldots \rightarrow T_1$ is maximal if there is no $T_{k+1}$ such that $T_k \rightarrow T_{k+1}$ or $T_k \rightarrow \ldots \rightarrow T_1$.

Similarly, $t_k \rightarrow \ldots \rightarrow t_1$ means $t_k$ references $t_{k-1}, t_{k-2}, \ldots, t_2$ references $t_1$.

**Definition 2**: A tuple $t_j \in T_1$ is a root of $T_k \rightarrow \ldots \rightarrow T_1$ if there are tuples $t_2 \in T_2, \ldots, t_k \in T_k$ such that $t_k \rightarrow \ldots \rightarrow t_1$. Let $S_{j,i}$ be the set of roots of $T_j \rightarrow \ldots \rightarrow T_1$, and $h_{j,i} = |S_{j,i}|$.

In Fig. 9, $a_2$ is a root of $T_C \rightarrow T_B \rightarrow T_A$, but not a root of $T_D \rightarrow T_C \rightarrow T_B \rightarrow T_A$. If $T_1 = T_A, T_2 = T_B, T_3 = T_C$ and $T_4 = T_D$, then $S_{4,2} = \{b_4, b_5\}$, so $h_{4,2} = 2$. There are 2 roots $a_2, a_3$ for $T_C \rightarrow T_B \rightarrow T_A$, and 3 roots $a_1, a_2, a_3$ for $T_B \rightarrow T_A$, so $h_{3,1} = 2, h_{2,1} = 3$.

The $h_{j,i}$ values form a matrix, as follows:

**Definition 3**: For a maximal chain $T_k \rightarrow \ldots \rightarrow T_1$, define
its **linear join matrix** as a lower triangular matrix $H$.

$$H = \begin{bmatrix}
0 & 0 & \cdots & 0 \\
h_{2,1} & h_{3,1} & \cdots & h_{k,1} \\
h_{3,2} & h_{4,2} & \cdots & h_{k,2} \\
\vdots & \vdots & \ddots & \vdots \\
h_{k,1} & h_{k,2} & \cdots & h_{k,k-1} \\
0 & 0 & \cdots & 0
\end{bmatrix}$$

Here, the linear join matrix defines the linear join property.

Let $H$ be a linear join matrix in some $\mathcal{D}_1$ before being tweaked by $\mathcal{T}_{\text{linear}}$, and $\tilde{H}$ be the target linear join matrix. The following theorem states necessary and sufficient conditions for $\mathcal{T}_{\text{linear}}$ to tweak $H$ to give $\tilde{H}$ (the proof is in the Appendix X-A):

**Theorem 1**: Let $H$ be the linear join matrix of $T_k \to \cdots \to T_1$ before being tweaked by $\mathcal{T}_{\text{linear}}$, and $\tilde{H}$ be the target linear join matrix. $H$ can be tweaked to $\tilde{H}$ if and only if

1. $\tilde{h}_{j,i} \leq \min_{n \leq n \leq j} |T_n|$ for all $1 \leq i < j \leq k$.
2. $\tilde{h}_{j,i} \geq \tilde{h}_{i+2,j} \geq \cdots \geq \tilde{h}_{k,i}$ for all $1 \leq i < k - 1$.
3. $\tilde{h}_{j,1} \leq \tilde{h}_{j,2} \leq \cdots \leq \tilde{h}_{j,j-1}$ for all $2 \leq j < k$.
4. $\tilde{h}_{j,i+1} \geq \tilde{h}_{j,i} - \tilde{h}_{j,j}$ for all $1 \leq i < j - 1 < k - 1$.

**B. Coappear property**

Fig. 10 illustrates the concept of a **coappear** property. 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. The same $\langle \text{postID}, \text{userID} \rangle$ may appear multiple times — i.e. postID and userID coappear — in the same table and in multiple tables.

This **coappear** property can be used for user profiling; e.g. if Alice comments, shares and likes a post about volunteerism many times, it is more likely that Alice is interested in volunteer work [30]. Other examples include group theme prediction [11] and on-line recommendation [19].

**Definition 4**: Suppose $T_1, \ldots, T_k$ reference the same tables $T_1', \ldots, T_m'$, and $b_1, \ldots, b_m\text{ coappear as foreign keys } v_1 \text{ times in } T_1', \ldots, v_k \text{ times in } T_k$. 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. 10, $\langle k_1, h_2 \rangle$ appears 3 times in $T_A$, 3 times in $T_B$, and 1 time in $T_C$, so $\xi(3,3,1) = 1$. Further, $\langle k_2, h_3 \rangle$ and $\langle k_3, h_1 \rangle$ each appears 1 time in $T_A$, 1 time in $T_B$ and 2 times in $T_C$, so $\xi(1,1,2) = 2$.

For the purpose of tweaking, the coappear distribution defines the coappear property. We have implemented a tool $\mathcal{T}_{\text{coappear}}$ to enforce this property, and included it in ASPECT. Like for linear joins, the following theorems state necessary and sufficient conditions for tweaking the coappear distribution, (the proofs can be found in the Appendix X-B):

**Theorem 2**: [**necessity**] Suppose tables $T_1, \ldots, T_k$ reference the same tables $T_1', \ldots, T_m'$ in some $\mathcal{D}_1$. $\xi$ is the coappear distribution before being tweaked by $\mathcal{T}_{\text{coappear}}$, and $\tilde{\xi}$ is the target coappear distribution. $\xi$ can be tweaked to $\tilde{\xi}$ only if:

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

\[\sum_v \tilde{\xi}(v) = m \prod_i |T_i'|\]

**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 in some $\mathcal{D}_1$ before tweaking and $\xi$ the target coappear distribution. If $\xi$ satisfies the necessary conditions in Theorem 2, then $\mathcal{T}_{\text{coappear}}$ tweaks $\xi$ to become $\tilde{\xi}$.

A dataset may have multiple coappear distributions. Suppose $T_D$ and $T_E$ reference $T_A$ and $T_B$, while $T_G$ and $T_H$ reference $T_B$ and $T_C$, so there are two coappear distributions: $\xi_{T_D,T_E}$ and $\xi_{T_G,T_H}$. $\mathcal{T}_{\text{coappear}}$ only modifies the referencing tables; e.g. tweaking $\xi_{T_D,T_E}$ 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.

However, in general, tools for tweaking coappear distributions can interfere with each other.

**C. Pairwise property**

Fig. 11 illustrates the concept of a **pairwise** property. In social networks, the social tie between users $u_1$ and $u_2$ may be implicit, instead of explicitly declared (as friends, say). For example, $u_1$ may respond twice $(r_1, r_2)$ to a post $p_3$ by $u_2$, whereas $u_2$ responds 4 times $(r_3, r_4, r_5, r_6)$ to 2 posts $p_1$ and $p_2$ by $u_1$. Such a property highlighted by previous work [33] 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 5:** Let R be a response2post table, and x, y ∈ \{0, 1, 2, ..., \}. Suppose there are k user pairs \( (u_1, v_1), \ldots, (u_k, v_k) \), 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.

Again, for the purpose of tweaking, a pairwise distribution defines the pairwise property. ASPECT currently have a \( T_{\text{pairwise}} \) that we have implemented for pairwise tweaking. Like for \( T_{\text{linear}} \), the following theorems state necessary and sufficient conditions for tweaking pairwise distributions (the proofs can be found in the Appendix X-C):

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

\[
\begin{align*}
(P1) \tilde{\rho}_R(x, y) &= \rho_R(y, x) \text{ for all } x, y \\
(P2) \sum_{x,y} (x+y)\tilde{\rho}_R(x, y) &= 2|T_R| \\
(P3) \sum_{x,y} \tilde{\rho}_R(x, y) &= |U|(|U| - 1) \quad \text{where } U \text{ is the user table.}
\end{align*}
\]

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

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 \( \rho_{R_i} \) is tweaked to \( \tilde{\rho}_{R_i} \) first. When tweaking \( \rho_{R_2} \), we only modify the tuples in \( R_2 \), so it does not affect the tweaked \( \tilde{\rho}_{R_1} \). Moreover, adding tuples in \( P \) does not affect \( \tilde{\rho}_{R_2} \) as well.

Again, in general, the tools for tweaking two pairwise distributions can interfere with each other.

In our implementation, the above-mentioned tweaking tools modify the dataset by calling the functions in Sec. III-D. Specifically, \( T_{\text{linear}} \) modifies the dataset through the operation replaceValues. \( T_{\text{coappear}} \) and \( T_{\text{pairwise}} \) modify the dataset through the operations deleteValues, insertValues.

VI. EXPERIMENT SETUP

We now describe the experiments for testing the effectiveness and efficiency of our \( T_{\text{linear}}, T_{\text{coappear}} \) and \( T_{\text{pairwise}} \) implementation. Each experiment is run on a Linux machine with 64GB memory and an Intel Xeon 2.4GHz processor. ASPECT is implemented in Java.

A. Datasets

Given the space constraint, this paper only presents experiments on Xiami\(^2\). The reader can refer to the Appendix for experiments on three other datasets, DoubanBook, DoubanMusic and DoubanMovie.

Xiami contains music-related data with 28 tables and more than 90M tuples. 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. (The Appendix X-D has more details about the datasets.)

We take 6 chronological 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 properties. After the tweaking process is done, ASPECT outputs a \( D_i \) that is similar to \( D_i \).

For our experiments, we use \( D_i \) as the ground truth, and compare the similarity between \( D_i \) and \( D_i \).

B. Size-scaler

Size-scalers are orthogonal to enforcing properties in the final dataset \( \tilde{D} \). Our experiments show that ASPECT is able to generate datasets with small errors for three different size-scalers, D scaler [37], ReX [8] and Rand, described below:

D scaler 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 properties do not satisfy the necessary conditions in Sec. V for datasets generated by ReX. We, therefore, modify the targeted properties to enforce the necessary conditions before tweaking.

Rand is a randomized size-scaler that generate the tuples randomly. However, it satisfies two requirements: (i) it generates the expected number of tuples, and (ii) the generated tuples satisfy the foreign key constraints.

C. Similarity measures

As stated previously, the similarity between tweaked dataset \( \tilde{D} \) and ground truth dataset \( D \) are defined through the property set II. Hence, we measure how well ASPECT preserves these properties. In the experiments, we only apply the 3 tweaking tools \( T_{\text{linear}}, T_{\text{coappear}} \) and \( T_{\text{pairwise}} \), and we use the corresponding properties to measure the similarity.

\(^2\)https://www.xiami.com
1) **Property accuracy**: We first measure the similarity of the 3 properties individually.

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

\[
\tilde{H} = \begin{bmatrix} 0 & 0 & 0 \\ 5 & 0 & 0 \\ 2 & 3 & 0 \end{bmatrix} \quad \text{and} \quad H = \begin{bmatrix} 0 & 0 & 0 \\ 4 & 0 & 0 \\ 3 & 4 & 0 \end{bmatrix},
\]

then \( \epsilon_H = \frac{1}{3}(\frac{5-4}{4} + \frac{2-3}{3} + \frac{3-4}{4}) = \frac{5}{12}. \) The linear property error of \( \tilde{D} \) is the mean of all \( \epsilon_H \), so it is unbounded.

**Coappear Property**: 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 property error of \( \tilde{D} \) is the mean of all \( \epsilon_{\xi} \).

**Pairwise Property**: For each 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 query \( q \) (e.g. \( \text{COUNT} \), \( \text{AVERAGE} \)) that are related to the 3 properties. The query error is measured by \( \epsilon_q = \frac{|q(D) - q(\tilde{D})|}{q(D)} \).

**VII. RESULTS AND ANALYSIS**

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

In the following, we compare property similarity in Sec. VII-A, and query similarity in Sec. VII-B. Sec. VII-C discusses possible improvements. Lastly, Sec. VII-D examines the execution time for these tools in ASPECT.

### A. Property similarity

For each property, each plot has an x-axis for the dataset snapshots, and a y-axis for the property error. In each plot, we compare how the 6 permutations perform against the baseline (i.e. without tweaking).

1) **Linear property**: In Xiami, there are 38 linear join matrices altogether, and Fig. 12 plots the average error for these 38 possibilities. In general, the later \( T_{\text{linear}} \) is applied, the smaller the linear property 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. The tweaking reduces the error tremendously for all tweaking orders, all size-scalers and all datasets.

Different size-scalers generate a scaled dataset with different errors. In Fig. 12, take \( D_0 \) 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.

2) **Coappear property**: There are 12 coappear distributions for Xiami, and Fig. 13 plots the average coappear error for these distributions. It shows that, like for \( T_{\text{linear}} \), the later \( T_{\text{coappear}} \) is 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 property, 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 that the original error is small and 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. III-C. We will discuss how to improve the similarity for such highly overlapping properties in Sec. VII-C.

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

3) Pairwise property: Xiami has 4 pairwise distributions, and Fig. 14 presents the pairwise error for tweaking these distributions. It again shows that, the later \( T_{\text{pairwise}} \) is applied in a tweaking order, the smaller the pairwise 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 property \( \Pi_i \). Moreover, all tweaking permutations reduce the errors tremendously for most of the cases. If the properties are highly overlapping, it is possible that the error reduction is not very significant. In Sec. VII-C, we will discuss potential improvements.

B. Query similarity

For each query \( q \), we compare the query results on the ground-truth dataset \( q(D) \) and the scaled dataset \( q(\bar{D}) \). As mentioned in Sec. VI-B, ReX cannot scale the dataset to arbitrary sizes, so there is no ground truth for its expected query results. We therefore omit ReX here.

We use 4 queries: \( Q_1 \) counts the number of users who have uploaded a photo with commenters; \( Q_2 \) counts 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; \( Q_4 \) counts the number of user pairs having interactions through profile page.

Fig. 15 presents results for these 4 queries. The first row uses Dscaler as a size-scaler, the second row uses Rand as a size-scaler. Each plot has an x-axis for the dataset snapshots, and a y-axis for query error.

As we can see from Fig. 15, all tweaking permutations reduce the query error significantly for both size-scalers. The errors are reduced to less than 0.05 for most of the tweaking 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 tweaking significantly reduces the errors for most of the cases, there are some rare exceptions, e.g. Dscaler-Xiami in Fig. 13, where ASPECT generates a dataset with coappear error greater than 0.1. Such cases happen when tools modify previously tweaked properties.

To improve the performance, we run ASPECT for more iterations. Previously, we applied tools \( T_{\text{coappear}}, T_{\text{linear}}, T_{\text{pairwise}} \) sequentially for the permutation C-L-P which results in \( D \). Now, we apply tools \( T_{\text{coappear}}, T_{\text{linear}}, T_{\text{pairwise}} \) on \( D \) in the same order with another few iterations. 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. 16 presents the results of C-L-P and C-P-L with more iterations. The x-axis is for properties, the y-axis is for errors, and the bars represent 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 property, the error reduction is greater, from 0.05 to \( 10^{-3} \) from third iteration onwards. For C-P-L, we observe a similar phenomenon, but the error reduction is faster: the error stabilises from the second iteration onwards.

In summary, the errors are reduced significantly as the number of iterations increases. From the second or third
iteration onwards, the errors are really small (about 0.02 or less), so the room for improvement becomes limited. The reader can find significant error reduction for other datasets in the Appendix X-E well.

D. ASPECT execution time

So far, we have verified that ASPECT is effective in tweaking the properties. We now show that ASPECT is efficient as well.

Fig. 17 presents the execution time of each tweaking permutation. Similar to previous plots, the x-axis represents the dataset snapshots, and the y-axis represents execution time (minutes).

In Fig. 17, the execution time increases linearly with the dataset size for most of the experiments. All tweaking permutations finish within 100 minutes. Different size-scalers have different property errors, so the amount of tweaking is different. It follows that different size-scalers have different execution times.

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.

VIII. LIMITATIONS AND OBSERVATIONS

ASPECT aims to tweak the properties \(\pi_1, \pi_2, \ldots, \pi_n\) so that the tweaked dataset has the corresponding target properties \(\bar{\pi}_1, \bar{\pi}_2, \ldots, \bar{\pi}_n\). Extensive experiments above show that ASPECT has the capability of tweaking complex properties with reasonably small errors within reasonable execution time.

A. Limitations

While tweaking a property, a tool may modify some already tweaked properties. These previous properties may take various forms, which significantly increases the difficulty of proving some error bounds for them. We formulate this issue as the Property Tweaking Bound Problem:

Assuming a dataset \(D_n\) has properties \(\bar{\pi}_1, \ldots, \bar{\pi}_n\). If a tweaking tool \(T_{n+1}\) is applied on \(D_n\), how much does it affect the previous properties \(\bar{\pi}_1, \ldots, \bar{\pi}_n\)?

Solving the Property Tweaking Bound Problem for general properties may not be possible. Proving error bounds should be easier if the properties satisfy certain requirements. Consider the following trivial example: if each \(\bar{\pi}_i\) represents the attribute distribution of a different column \(C_i\), then one can easily see that tweaking tool \(T_{n+1}\) will never affect \(\bar{\pi}_1, \ldots, \bar{\pi}_n\).

The following theorem shows what we can prove for a non-trivial restriction, if all the properties refer to just 1 column:

Theorem 6: If \(\pi_1, \ldots, \pi_{n+1}\) are the frequency distributions of the same column, and we have already applied tools \(T_1, \ldots, T_n\) which gives us \(\bar{D}_n\). If a tweaking tool \(T_{n+1}\) is applied on \(\bar{D}_n\), then the total error is \(\sum_{1 \leq i \leq n} ||\bar{\pi}_i - \pi_{n+1}||\), where \(||\bar{\pi}_i - \pi_{n+1}||\) is the frequency difference between \(\pi_i\) and \(\pi_{n+1}\). (The proof is provided in the Appendix X-H1)

For example, \(\bar{\pi}_1\) and \(\bar{\pi}_2\) both represent the properties on the age column. Here, \(\pi_1(10) = 20\%\) means there are 20% of the population at the age of 10. Then, difference between \(\bar{\pi}_1\) and \(\bar{\pi}_2\) is \(\sum ||\bar{\pi}_1(i) - \pi_2(i)||\). By extending Theorem 6, we have the following Theorem for properties that overlap on one column.

Theorem 7: If \(\pi_1, \ldots, \pi_{n+1}\) are the frequency distributions over multiple columns and sharing the one common column \(c_0\), and we have already applied tools \(T_1, \ldots, T_n\) which gives us \(\bar{D}_n\). If a tweaking tool \(T_{n+1}\) is applied on \(\bar{D}_n\), then the total error is \(\sum_{1 \leq i \leq n} ||\pi_i - \pi_{n+1}||_{c_0}\), where \(||\pi_i - \pi_{n+1}||_{c_0}\) is the frequency difference between \(\pi_i\) and \(\pi_{n+1}\) over column \(c_0\). (The proof is provided in the Appendix X-H2)

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

When tweaking a dataset for \(n\) properties, which tweaking order results in the least error? Similarly, we have the following theorem:

Theorem 8: If \(\pi_1, \ldots, \pi_n\) are frequency distributions for the same column, and \(\bar{\pi}_{\text{min}}\) is the property having the minimum total amount of differences with \(\pi_1, \ldots, \pi_n\). Then, when tweaking a dataset for these \(n\) properties, the total error is minimized if \(T_{\text{min}}\) is the last in the tweaking order. (The proof is provided in the Appendix X-H3)

We believe the Property Tweaking Bound Problem and Property 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 properties. In general, we say two properties overlap if they refer to the same attribute value in a tuple. If the property \(\pi_{n+1}\) does not overlap with all previous properties, then regardless of the modification of \(T_{n+1}\), all the previous modified properties will be preserved.

(O2) Determination of non-overlapping properties. Given (O1), we would want to determine those properties among \(\pi_1, \ldots, \pi_n\) that do not overlap. Then, the user will clearly know those properties do not affect each other. This can be achieved in ASPECT through monitoring the dataset access by each tweak. In ASPECT, each tweak can only access the dataset via functions similar to the ones provided in Fig. 5. Hence, ASPECT knows if any two tweaking tools...
have accessed the same tuples. We can then construct a graph with tools as nodes, and have an edge between two tools that access the same tuples. The problem is thus reduced to finding independent sets in graph theory. Although it is NP-Hard to find a maximum independent set, Robson [29] has proven that it can be done in $O(1.22^n)$ time, which is a reasonable complexity if the number of tools ($n$) is small.

**O3 Conflicting overlapping properties.** We say overlapping properties are conflicting if no dataset can satisfy all of them. A simple example of conflicting properties of a social network dataset is: $F_1 = \{\text{more than half the users are men}\}$ and $F_2 = \{\text{more than half the users are women}\}$. Such properties must be modified to resolve the conflict, and ASPECT modifies the properties that are applied earlier.

**O4 Non-conflicting overlapping properties.** For non-conflicting properties, 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-time algorithm to generate a graph for such distributions. For the sake of efficiency, we may have to sacrifice some property accuracy. Even so, for the properties in this paper, ASPECT enforces the properties accurately and efficiently. As we can see from Sec. VII-C, the error is reduced to 0.02 after 2 to 3 iterations in the experiments.

**IX. CONCLUSION**

This paper introduces ASPECT, a framework with

- a repository for developers to contribute tools that tweak a dataset to enforce desired properties, and
- a uniform interface for the tools to collaborate and interoperate.

To generate a scaled dataset with greater similarity to the original dataset, one just needs to apply more tweaking tools.

We demonstrate ASPECT with 3 highly overlapping and nontrivial properties, and apply their corresponding tools on real datasets. Extensive experiments show that ASPECT effectively reduces the errors by orders of magnitude in the synthetic data without sacrificing efficiency.

ASPECT is a step towards the vision for application-specific benchmark data generation. It facilitates collaboration among developers in contributing tools for tweaking synthetic datasets to enforce similarity with empirical data.

**REFERENCES**


X. APPENDIX

A. Necessity and sufficiency proof of Theorem 1

1) Necessity proof of Theorem 1: Proof: (L1) The condition says the number of roots is not more than any table size along the linear join. Consider the directed trees defined by the tuple references, like in Fig. 9. Since each tuple has at most one parent, \( |T_n| \) is at least the number of roots \( h_{j,i} \) for any \( i \leq n \leq j \).

(L2) The condition says the elements in \( \bar{H} \) for a column are non-increasing. It follows from observing that every path from \( T_{j+1} \) to \( T_i \) contains a sub-path from \( T_j \) to \( T_i \).

(L3) This condition says the elements in \( \bar{H} \) for a row are non-decreasing. It follows from observing that every path from \( T_j \) to \( T_i \) contains a sub-path from \( T_j \) to \( T_{i+1} \).

(L4) Since \( S_{j+1,i} \subseteq S_{j,i} \), then \( h_{j,i}−h_{j+1,i} = |S_{j,i}|−|S_{j+1,i}| \leq S_{j,i}−S_{j+1,i} \). Then, \( h_{j,i+1}−h_{j+1,i+1} = |S_{j,i+1}−S_{j+1,i+1}| \. Any \( t \in S_{j,i}−S_{j+1,i} \) has a child \( t' \) that has a path from \( T_j \) but not from \( T_{j+1} \), so \( t' \in S_{j,i+1}−S_{j+1,i+1} \). Thus \( |S_{j,i}−S_{j+1,i}| \leq |S_{j,i+1}−S_{j+1,i+1}| \) and (L4) follows.

2) Sufficiency proof of Theorem 1: Now, we describe how \( T_{\text{linear}} \) tweaks \( H \) to \( \bar{H} \). \( T_{\text{linear}} \) tweaks \( H \) row by row. For the first row, \( T_{\text{linear}} \) then tweaks the row entry by entry. \( T_{\text{linear}} \) first does \( \text{leadingAdjust} \), tweaking \((h_{1,1}, h_{1,2}, \ldots, h_{1,i−1}) \) to \((\tilde{h}_{1,1}, \ldots) \). Then, \( T_{\text{linear}} \) then does \( \text{nonLeadingAdjust} \): it tweaks \((h_{i,1}, \ldots) \) to \((\tilde{h}_{i,1}, \tilde{h}_{i,2}, \ldots) \) to \((\tilde{h}_{i,1}, \tilde{h}_{i,2}, \tilde{h}_{i,3}, \ldots, h_{i,i−1}) \). Instead of providing a formal proof, we present an example of tweaking from \( H_{\text{exp}} \) to \( H_{\text{exp}} \) in Fig. 18, then followed by the full proof.

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 \( (b) \) 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.

Algorithm 1: Linear Property Tweaking

input : \( H, H, D \)
1 for each row of \( H \) do
2 leadingElementAdjust() //Lemma 1
3 for 2 to row_length do
4 nonLeadingElementAdjust() //Lemma 3

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.

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 \( c_2 \). 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 \( T_D \rightarrow T_C \rightarrow T_B \rightarrow T_A \). 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 4th 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.

In this section, we present the the formal proofs that are related to the linear property. Define \( H^* = H−\bar{H} \) and \( h_{j,i} = h_{j,i}−\bar{h}_{j,i} \) for all \( 1 \leq i, j \leq k \). Here, \( H \) is the linear join matrix of the chain \( T_K \rightarrow \cdots \rightarrow T_1 \).

Lemma 1 (leadingElementAdjust): If the first \( n \) rows of \( H \) and \( \bar{H} \) are the same, then \((h_{n+1,1}, h_{n+1,2}, \ldots, h_{n+1,n}) \) can be tweaked to \((0, h_{n+2,2}, \ldots, h_{n+1,n}) \), where \( h_{n+1,i} \geq 0 \) 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 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 $T_n$ 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 $Leaf_{n+1,1}$. It is obvious that plucking any tuples from $T_{n+1} - Leaf_{n+1,1}$ never modify $h_{n+1,1}^*$. Then,

$$|T_{n+1} - Leaf_{n+1,1}| - |h_{n+1,1}^*|$$

$$= |T_{n+1}| - h_{n+1,1} - (h_{n+1,1} - h_{n+1,1})$$

$$= |T_{n+1}| - h_{n+1,1} + 0 \quad (By \ L1)$$

Hence, $|h_{n+1,1}^*|$ tuples in $T_{n+1} - Leaf_{n+1,1}$ 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 $T_n$ having descendants in $T_n$ but no descendants in $T_{n+1}$. Since

$$|S_{n+1} - S_{n+1,1}| - |h_{n+1,1}^*|$$

$$= |h_{n+1,1} - h_{n+1,1}^*| - (h_{n+1,1} - h_{n+1,1})$$

$$= h_{n+1,1} - h_{n+1,1} + 0 \quad (By \ L2)$$

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, $h_{n+1,1,i}^* = 0$ after attachment.

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

Let’s explain with examples in Figure 19 first. Suppose $h_{n+1,2}^* = 1$, so we want to remove descendants in $T_D$ for 1 tuple in $T_B$. If we do this by plucking $d_0$ from $c_5$, then $h_{4,1}^*$ is decreased by 1 as well, so we should instead pluck $d_0$ or $d_2$. Therefore, when tweaking $h_{n+1,1}^*$, we should avoid affecting $h_{n+1,1,i}^*$. 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 it to $c_6$; this will increase $h_{4,2}$, but it will also increase $h_{4,1}$. However, if we first pluck the subtree rooted at $b_0$ and attach it to $a_2$, then pluck $d_1$ and attach it to $c_6$, $h_{4,2}$ will increase without affecting other $h_{j,i}$ values. This leads us to the following definition:

**Definition 6:** For a reference chain $T_k \rightarrow \cdots \rightarrow T_1$, suppose we pluck $t \in T_1$ from $t' \in T_{i-1}$ and attach $t$ to some other $t'' \in T_{i-1}$. We call this an **isomorphic adjustment** if the linear join matrix is unchanged.

**Lemma 2:** For a reference chain $T_k \rightarrow \cdots \rightarrow T_1$, we can make $|S_{k-1,i} - S_{k,i}| - |S_{k-1,i-1} - S_{k,i-1}|$ isomorphic adjustments to $T_i$.

**Proof:** $(S_{k-1,i-1} - S_{k,i-1})$ are the tuples in $T_{i-1}$ having descendants in $T_{k-1}$ but no descendants in $T_k$. Similarly, $(S_{k-1,i} - S_{k,i})$ are the tuples in $T_i$ having descendants in $T_{k-1}$ but no descendants in $T_k$. For each tuple in $S_{k-1,i-1} - S_{k,i-1}$, pick 1 descendant in $T_i$ with descendants in $T_{k-1}$ to form a set $S_y$. Then, $S_y \subseteq S_{k-1,i} - S_{k,i}$. Let’s consider $S_{k-1,i} - S_{k,i} - S_y$. For any tuple in $S_{k-1,i} - S_{k,i} - S_y$, we can pluck it and attach it back randomly to tuples in $S_{k,i}$. These adjustments are isomorphic. Hence, the maximum number is $|S_{k-1,i} - S_{k,i} - S_y| = |S_{k-1,i} - S_{k,i} - (|S_{k-1,i} - S_{k,i}-1)|$.

**Lemma 3 (nonLeadingElementAdjust):**

Suppose the first $n$ rows of $H$ and $\tilde{H}$ are the same. Then $(0, \ldots, 0, h_{n+1,i}^*, \ldots, h_{n+1,n}^*)$ can be tweaked to $(0, \ldots, 0, h_{n+1,i+1}^*, \ldots, h_{n+1,n}^*)$.

**Proof:** The proof and tweaking steps are similar to Lemma 1. There are two cases:

**Case $h_{n+1,i}^* > 0$: Similar to Lemma 1. There are two steps:**

**Leaf Tuple Plucking:** For each tuple in $S_{n+1,i} - S_{n+1,i-1}$, pick one descendant $t_y$ in $T_i$, where $t_y$ has descendants in $T_{i+1}$. Use $R_y$ to denote the set of all such $t_y$. Therefore, $|R_y| = |S_{n+1,i-1}|$. For any tuple in $S_{n+1,i} - R_y$, $h_{n+1,i}$ will be decreased if all its descendants in $T_{n+1}$ are detached. Moreover, such detachment will not affect $h_{n+1,i-1}$. Thus,

$$|S_{n+1,i} - R_y| = h_{n+1,i} - h_{n+1,i-1}$$

$$\geq h_{n+1,i} - h_{n+1,i} \quad (By \ L3)$$

Then, $h_{n+1,i}$ tuples in $S_{n+1,i} - R_y$ can be randomly plucked.

**Leaf Tuple Attaching:** Similar to Lemma 1.

**Case $h_{n+1,i}^* < 0$: Similar to Lemma 1, there are two steps:**

Leaf Tuple Plucking and Leaf Tuple Attaching.

**Leaf Tuple Attaching:** Similar to Lemma 1.
Moreover,
\[ |S_{n,i}| - |S_{n+1,i}| - (|S_{n,i-1}| - |S_{n+1,i-1}|) = |h_{n+1,i}^*| \]
\[ = h_{n,i} - h_{n+1,i} + h_{n+1,i-1} - h_{n,i-1} + \bar{h}_{n+1,i} + h_{n+1,i} \]
\[ = h_{n,i} + h_{n+1,i-1} - h_{n,i-1} + \bar{h}_{n+1,i} \]
\[ = \bar{h}_{n,i} + \bar{h}_{n+1,i} - h_{n,i-1} + \bar{h}_{n+1,i} \geq 0 \] (By LA)

Therefore, at least \( |h_{n+1,i}^*| \) isomorphisms 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.

**Theorem 9:** For a reference chain \( T_k \rightarrow \cdots \rightarrow T_1 \) in some \( D_i \), let \( H \) be the linear join matrix before tweaking and \( \tilde{H} \) the target linear join matrix. If \( H \) satisfies the necessary conditions in Theorem 1, then Algorithm 1 tweaks \( H \) to give \( \tilde{H} \); for each row, the first entry is tweaked to 0 with Lemma 1, and the following entries are tweaked to 0 with Lemma 3.

**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 withLemma 3.

**B. Necessity and sufficiency proof for coappear property**

**Algorithm 2:** Coappear Property Tweeping

input: \( \xi, \xi', \tilde{D} \)

1. for each \( (v_1, \ldots, v_k) \in \Delta^- \) do
   2. while \( \xi^*(v_1, \ldots, v_k) \neq 0 \) do
      3. \( (v'_1, \ldots, v'_k) \leftarrow \text{closest}(\Delta^+, (v_1, \ldots, v_k)) \)
      4. for \( i \in [1, k] \) do
         5. \( t_i \leftarrow \text{tupleRetrieve}(T_i, (v'_1, \ldots, v'_k)) \)
         6. \( \text{tupleModification}(T_i, v'_i - v_i, t_i) \)
      7. \( \text{statsUpdate}(\xi^*(v_1, \ldots, v_k), \xi^*(v'_1, \ldots, v'_k)) \)

1) **Necessity proof of Theorem 2:** Proof: (C1) \( \tilde{\xi}^0(v) \) 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 \( \sum v_i \xi(v) = |T_i| \).

(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 \( \prod_{i=1}^m |T_i| \).

2) **Tweaking algorithm for coappear property:** Next, we explain how \( T_{\text{coappear}} \) tweaks \( \xi \) to \( \xi' \). Let \( \xi = \xi' + \tilde{\xi} \). \( \Delta^* = \{ \xi' \geq 0 \} \). \( \Delta^- = \{ \xi' > 0 \} \) and \( \Delta^+ = \{ \xi' = 0 \} \). \( T_{\text{coappear}} \) 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. 10, \( (k_2, h_3) \) and \( (k_3, h_3) \) both have \( v' = (1, 1, 2) \)). For each \( v' \), choose one such \( b \).

**Tuple 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 now prove that the necessary conditions are sufficient for the tweaking, which is Theorem 3.

**Proof:** Both \( \xi \) and \( \tilde{\xi} \) satisfy C1, so \( |T_i| \) is unaffected by the tweaking. Similarly, C2 ensures \( \sum v_i \xi = \sum v_i \xi' \), so \( \sum v_i \xi = 0 \). Therefore

\[ \sum_{v \in \Delta^+} \xi^*(v) + \sum_{v \in \Delta^-} \xi^*(v) + \sum_{v \in \Delta^-} \xi^*(v) = 0, \]

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 \( \xi = 0 \); i.e. \( \xi = \xi' \).

**C. Necessity and sufficiency proof for pairwise property**

1) **Necessity proof for Theorem 4:** Proof: (P1) \( \rho_R(x, y) = k \) means there are \( k \) user pairs \( (u_i, v_i) \), 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 \( \text{response2post} \) for each \( (u_i, v_i) \). By symmetry, these \( x + y \) tuples also represent \( k \) \( (v_i, u_i) \) pairs, so \( \rho_R(y, x) = k \).

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

(P3) Similarly, there are \( |U|(|U| - 1) \) user pairs, and each is counted once by \( \rho_R(x, y) \), so (P3) follows.

2) **Tweaking algorithm for pairwise property:** Next, we explain how \( T_{\text{pairwise}} \) tweaks \( \rho \) to \( \tilde{\rho} \). Let \( \rho_R = \rho_R - \rho_R - \rho_R, \Theta^+ = \{ (x, y) | \rho_R(x, y) > 0 \} \) and \( \Theta^- = \{ (x, y) | \rho_R(x, y) < 0 \} \). \( T_{\text{pairwise}} \) loops through each \( \text{response2post} \) table \( R \). For each \( (x, y) \in \Theta^- \), it adds \( |\rho_R(x, y)| \) pairs \( (u_i, v_i) \), where user \( u_i/v_i \) has \( x/y \) \( \text{response2post} \) tuples in \( R \) referencing \( v_i/u_i \)'s post. It does this by looping \( |\rho_R(x, y)| \) times, and in each iteration:

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

- **TupleModification:** Choose users \( u_i/v_i \) with pairwise vector \( (x', y') \) and tweak \( u_i/v_i \)'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 \( T_{\text{pairwise}} \) 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 \( p_i \) with minimum responses among \( w_i \)'s posts; we make \( p_i \) a post by \( v_i \), and shift the responses to \( p_i \) 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.
Algorithm 3: Pairwise Property Tweaking

input : $\rho, \tilde{\rho}, \tilde{D}$

1 for each $\rho^*_R$ do
2 for each $(x, y) \in \Theta^-$ do
3 \[ \text{while } \rho^*_R(x, y) \neq 0 \text{ do} \]
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 \[ \text{statsUpdate}(\rho^*_R(x', y')) \]

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.

Now we formally prove Theorem 5.

Proof: Since $\rho_R$ and $\tilde{\rho}_R$ both satisfy (P3), and $T_{\text{pairwise}}$ 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.

3) Relaxing the assumption: 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. Previous section has discussed the case for $\rho_N$, so we now discuss tweaking for $\rho_S$.

Theorem 10: For a response2post table $R$ in some $\tilde{D}_i$, let $\rho_S$ be the pairwise distribution generated by user self-responding behavior. If $\rho_S$ can be tweaked to become $\tilde{\rho}_S$, then $\tilde{\rho}_S$ satisfies the following conditions:

(SP1) $\sum_x 2x\tilde{\rho}_S(x, x) + \sum_{x,y} (x + y)|\tilde{\rho}_N(x, y)| = 2|T_R|$

(SP2) $\tilde{\rho}_S(x, x) = |U|$ where $U$ is the user table.

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 $\tilde{\rho}_N(x, y)$ and $\tilde{\rho}_N(x, y)$. 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 $(u, u_i)$, 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 pair-wise 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 $T_{\text{pairwise}}$ randomly chooses and removes $x' - x$ such responses. If $x > x'$: $T_{\text{pairwise}}$ 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 $w$. 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 10 suffices to ensure that $T_{\text{pairwise}}$ tweaks $\rho_S$ to become $\tilde{\rho}_S$.

Theorem 11: For each response2post table $R$ in some $\tilde{D}_i$, let $\rho_S$ be the self-responded pairwise distribution before tweaking and $\tilde{\rho}_S$ the target pairwise distribution. If $\tilde{\rho}_S$ satisfies the conditions in Theorem 10, then $\rho_S$ can be tweaked to $\tilde{\rho}_S$.

Moreover, the extra tuples added to the post table $P$ is at most $|U| - |P|$, where $U$ is the user table.

Proof: The proof is similar to the proof in Theorem 5.

D. Dataset summary

In this section, we summarize the datasets used in the experiments. We used 4 datasets from Douban$^3$ and Xiami$^4$. 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, 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.

$^3$https://www.douban.com
$^4$https://www.xiami.com
4) Xiami also contains music-related data, but is larger: It has 26 tables and more than 90millions tuples.

Figure 20 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.

**Fig. 20:** Dataset size summary

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$D_2$</th>
<th>$D_3$</th>
<th>$D_4$</th>
<th>$D_5$</th>
<th>$D_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xiami</td>
<td>208M</td>
<td>551M</td>
<td>881M</td>
<td>1.2G</td>
<td>1.9G</td>
</tr>
<tr>
<td>DoubanMovie</td>
<td>1.6G</td>
<td>1.8G</td>
<td>2.0G</td>
<td>2.2G</td>
<td>2.5G</td>
</tr>
<tr>
<td>DoubanMusic</td>
<td>348M</td>
<td>396M</td>
<td>454M</td>
<td>512M</td>
<td>570M</td>
</tr>
<tr>
<td>DoubanBook</td>
<td>646M</td>
<td>736M</td>
<td>845M</td>
<td>971M</td>
<td>1.1G</td>
</tr>
</tbody>
</table>

**Fig. 21:** Schema for DoubanMusic: There are 11 tables; The tables with the same color (except grey color) share the same coappear distribution. For example, Album_Comment, Album_Listening, Album_Heard, Album_Wish reference to both Album and User tables; Review is the post table; Review_Comment is the response2post table.

**Fig. 22:** Schema for DoubanBook: There are 12 tables; The tables with the same color (except grey color) share the same coappear distribution. For example, Book_Comment, Book_Reading, Book_Read, Book_Wish, Diary reference to both Book and User tables; Diary and Review are the post tables; Diary_Comment Review_Comment are the response2post tables.

**Fig. 23:** Schema for DoubanMovie: There are 17 tables; The tables with the same color (except grey color) share the same coappear distribution. For example, Movie_Actor, Movie_Script, Movie_Director reference to both Star and Movie tables; Movie_Review and Movie_Photo are the post tables; Review_Comment and Photo_Comment are the response2post tables.

**Fig. 24:** Schema for Xiami: There are 28 tables; The tables with the same color (except grey color) share the same coappear distribution. For example, Listen_Song, Lib_Song reference to both Song and User tables; Collection, Photo, Space and Thread are the post tables; Photo_Comment, Space_Comment, Collect_Like and Thread_Comment are the response2post tables.

**E. Similarity experiments for DoubanMovie, DoubanMusic, DoubanBook**

1) Property similarity for DoubanMovie, DoubanMusic, DoubanBook: In this section, we presents the property simi-
Figure 25 presents the linear property similarity results. All tables are involved in at least 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 property 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 property 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 Section VII-A, such highly overlapped properties increase the difficulty of getting a validated modification as described in Section III. Hence, this could be a potential reason that error is $> 0.1$.

Figure 26 presents the coappear property similarity results. There are 6 coappear distributions for DoubanMovie. For example, The 6 tables (Movie_Seen, Movie_Watching, Movie_Wish, Movie_Photograph, Movie_Review,
Movie_Comment) reference Movie and User. Similarly, DoubanMusic has 4 coappear distributions, DoubanBook has 4 and Xiami has 12. In each case, each table is involved in one or more coappear distributions.

Figure 26 shows that, like for $T_{\text{linear}}$, the latter $T_{\text{coappear}}$ 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 III. 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.

Figure 27 presents the pairwise property similarity results. DoubanMovie has 2 pairwise distributions: (i) Review as post table and Review_Comment as response2post table; and (ii) Photo as post and Photo_Comment as response2post. DoubanMusic has 1 pairwise distribution, and DoubanBook has 2 pairwise distributions.

Figure 27 again shows that, the later $T_{\text{pairwise}}$ is applied in a tweaking order, the smaller the pairwise property error in the tweaked dataset. For DoubanMusic and Xiami, all tweaking permutations reduce the errors tremendously for all size-scalers. For DoubanMovie, all tweaking permutations on data generated by Dscaler significantly reduce the pairwise property error; most tweaking permutations on data generated by Rand significantly reduce the error, except L-P-C and P-L-C. For Dscaler-DoubanMovie, the error without tweaking is small (< 0.05), some tweaking permutations increase the errors. For DoubanBook, all tweaking permutations reduce the errors tremendously for all size-scalers except three tweaking permutations on Dscaler-DoubanBook.

2) 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.

Figure 28 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.
In Figure 28, all tweaking permutations reduce the query error significantly. The errors are reduced to < 0.05 for most of the tweaking permutations.

Figure 29 presents the query results on DoubanMusic. 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 stars that have at most 10 different fans; $Q_3$ computes the average number of interested listeners of a book; $Q_4$ computes the number of user pairs having interactions through a book review.

Similar to DoubanMovie, all permutations reduce the errors tremendously.

Figure 30 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 number of users that have at most 10 different fans; $Q_4$ computes the number of user pairs having interactions through a book review.

As we can see from Figure 30, most of the tweaking permutations reduce the errors tremendously except for few rare cases, e.g., 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 property related query, and the linear property that were tweaked by $T_{linear}$ is subsequently modified by $T_{coappear}$ and $T_{pairwise}$. Such a scenario can be improved by having more iterations. In Figure 31, we run L-C-P on Dscaler-DoubanBook with more iterations. We can see that from second iteration onwards, the Q1 error is reduced to less than 0.001.

In this section, we present property similarity results for different iterations. Figure 32, Figure 33 and Figure 34 present the results of running 6 tweaking permutations for up to 4 iterations on the dataset generated by Dscaler, ReX and Rand. Take Figure 32 for example, for coappear property, 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 property 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.
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 property 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.

H. Proofs in Sec.VIII

1) Proof of Theorem 6: Proof: Let \( \pi_1, \ldots, \pi_{n+1} \) be the frequency distributions of the same column, where \( \pi_i(x) = p \) means \( p \) portion of the values in the column are expected to be \( x \). After \( T_{n+1} \) is applied on \( D_n \), then \( D_{n+1} \) has property \( \pi_{n+1} \). Hence, the error of \( \pi_i \) is \( \sum_x |\pi_i(x) - \pi_{n+1}(x)| \). Hence, the theorem is proven.

2) Proof of Theorem 7: Proof: Let \( \pi_1, \ldots, \pi_{n+1} \) be the frequency distributions over multiple columns and sharing one common column \( c_0 \), and \( \pi_{nc0} \) is the frequency distribution for column \( c_0 \) for property \( \pi_i \). After \( T_{n+1} \) is applied on \( D_n \), then \( D_{n+1} \) has property \( \pi_{n+1} \). From Theorem 6, we know that the error between \( \pi_{nc0} \) and \( \pi_{n+1c0} \) is \( \sum_x |\pi_{nc0}(x) - \pi_{n+1c0}(x)| \). Let \( \pi_i^* \) be the property that is presented in \( D_n \) for the targeting property \( \pi_i \). It is easy to see that \( \sum_0 |\pi_i(v) - \pi_i^*(v)| \geq \sum_x |\pi_{nc0}(x) - \pi_{n+1c0}(x)| \). Hence, the theorem is proven.

3) Proof of Theorem 8 : Proof: Let \( T_{j} \) be the tool applied last, from Theorem 6, we know that the error is \( \sum_i ||\pi_i - \pi_j|| \). Since \( \pi_{min} \) is the property which has the smallest error \( \sum_i ||\pi_i - \pi_{min}|| \). Hence, the error is minimized if \( T_{min} \) is the last applied tool.

Fig. 35: Execution time (x-axis represents the dataset snapshots; y-axis is the execution time in minutes; No-Tweak represents the experimental result without any tweaking; Dscaler - DoubanMovie represents using Dscaler as a size-scaler on DoubanMovie.)