SAM: Database Generation from Query Workloads with Supervised Autoregressive Models

Jingyi Yang[1]*, Peizhi Wu[2]*, Gao Cong[1], Tieying Zhang[3], Xiao He[4]
[1]Nanyang Technological University [2]University of Pennsylvania [3]ByteDance Inc. [4]Alibaba Group
{jyang028@e.,gaocong@}ntu.edu.sg,pagewu@cis.upenn.edu
tieying.zhang@bytedance.com,xiao.hx@alibaba-inc.com

ABSTRACT

With the prevalence of cloud databases, database users are increasingly reliant on the cloud database providers to manage their data. It becomes a challenge for cloud providers to benchmark different DBMS for a specific database instance without having access to the underlying data. One viable solution is to leverage a query workload, which contains a set of queries and the corresponding cardinalities, to generate a synthetic database with similar query performance. Existing methods for database generation with cardinality constraints, however, can only handle very small query workloads due to their high complexity and encounter challenges when handling join queries.

In this work, we propose SAM, a supervised deep autoregressive model-based method for database generation from query workloads. First, SAM is able to process large-scale query workloads efficiently as its complexity is linear in the size of the query workload, the number of attributes and the attribute domain size. Second, we develop algorithms to obtain unbiased samples of base relations from the deep autoregressive model and assign join keys in a way that accurately recovers the full outer join of the target database. Comprehensive experiments on real-world datasets demonstrate that SAM is able to efficiently generate a high-fidelity database that not only satisfies the input cardinality constraints, but also is close to the target database.

CCS CONCEPTS

• **Computing methodologies → Machine learning**; • **Theory of computation → Database constraints theory**; • **Information systems → Database management system engines**.

KEYWORDS

Database Generation, Supervised Autoregressive Models

ACM Reference Format:
Jingyi Yang[1]*, Peizhi Wu[2]*, Gao Cong[1], Tieying Zhang[3], Xiao He[4]. 2022. SAM: Database Generation from Query Workloads with Supervised Autoregressive Models. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526168

*Both authors contributed equally to this research.

1 INTRODUCTION

As cloud computing becomes prevalent, databases are also migrating to the cloud, providing scalability and reduced administrative burden for database users. Therefore, nowadays, an increasing number of individuals and enterprises use databases managed by cloud providers rather than having them on-premise. Before a user migrates the databases from local to cloud storage or from one cloud provider to another, the cloud provider may wish to recommend a DBMS product to the user based on the benchmark results of different DBMS on the user's databases and workloads.

Benchmarking requires access to the users' databases, which are not granted to the cloud provider before the actual migration. On the other hand, cloud providers may have access to the users' query workloads, which contain the queries and the result cardinalities, since query workloads have lower security level than the data. The query workloads can be considered as a set of cardinality constraints specifying the data characteristics of the underlying database [4]. Therefore, one natural idea to address the aforementioned issues is to leverage the query workloads to generate a database that satisfies the cardinality constraints as a fundamental to replay workloads [1]. Benchmarking can then be conducted on the generated database, which is an approximate of the original database.

Another typical use case of database generation is stress testing for databases with strict access controls. For enterprise database users like e-commerce and social media platforms, their engineers need to stress test the databases to ensure production stability under high traffic. However, replicating the entire database for stress testing is highly restricted for databases with a high security level and strict access control constraints, *e.g.*, core user database of a social network/e-commerce platform. Again, one can first query the target database and obtain the result cardinalities of a query workload, and generate a synthetic database from the query workload for the purpose of stress testing.

Existing work [4] on database generation with cardinality constraints uses Probabilistic Graphical Model (PGM) to construct a generative distribution of the database, which can then be sampled from to generate a synthetic database. Since query workloads are often of a large scale, it becomes possible to accurately model the joint distribution of a database by processing a large number of cardinality constraints. However, PGM-based approaches [4] have a complexity that scales quickly with regard to the number of correlated attributes and attribute domain size, both of which increase as the number of query constraints increases. In our experiments, it is only able to handle a small number of cardinality constraints for a single relation within a reasonable time frame (*e.g.*, 12 hours). Furthermore, it is even more challenging for PGM-based methods to handle multiple relations: they need to build separate models for queries involving different subsets of relations. As the models are built from disjoint sets of query constraints, there may exist inconsistencies across different models, reducing the fidelity of generated data for multi-relation databases. Detailed discussions of PGM-based methods can be found in Section 2.3.

To empower the possibility of processing query workloads of different scales while eliminating inconsistencies in the modeling of data distribution, we consider a new class of models for database generation – deep autoregressive models [12, 27]. Inspired by the recent advances in deep autoregressive models for cardinality estimation [34–36], we propose a **S**upervised deep **A**utoregressive **M**odel-based method for database generation — SAM. SAM first trains a single autoregressive model of the joint data distribution of the entire database from a query workload. The training process scales linearly with regard to the size of the input query workload and can be greatly accelerated with GPUs. With an autoregressive model of the joint data distribution, it is straightforward to generate a database instance of a single-relation. However, for multi-relation databases, it remains a huge challenge to generate the base relations from the deep autoregressive model, which will be discussed in Section 4.3. To address this, we further propose *inverse probability weighting* and *scaling* algorithms to obtain unbiased base relation samples from full outer join samples, and a novel *Group-and-Merge* algorithms to assign join keys to the generated relations so that they can better recover the full outer join, achieving high data fidelity.

Contributions This work makes the following contributions

- We design and implement SAM, a query-aware database generator based on autoregressive models. Unlike previous methods [4] with a high time complexity, SAM has a linear complexity with regard to the size of the query workload and is therefore capable of processing query workloads of a much larger scale, *i.e.*, 250×, within a fixed time frame. To the best of our knowledge, this is the first model that is able to generate databases satisfying a large number of cardinality constraints.
- SAM uses a single autoregressive model to handle single-relation, multi-relation and join queries, so that the modeled data distribution is consistent with all queries. We propose *inverse probability weighting* and *scaling* to efficiently sample from the single autoregressive model, and produce unbiased samples for each base relation.
- We propose a novel *Group-and-Merge* algorithm to assign join key to the generated base relations so that SAM is capable of recovering the full outer join of the target database.
- We conduct comprehensive experiments to show that SAM significantly outperforms the previous approach [4] in terms of data fidelity, database recovery, and efficiency.

2 PRELIMINARIES

In this section, we first define relevant notations in Section 2.1 and then define the problem of *Database Generation from Query Workloads* in Section 2.2. We discuss major previous work on this problem and its shortcomings in Section 2.3.

2.1 Notations

Consider a relation T that has n columns (or attributes) $\{A_1, A_2, ..., A_n\}$. The size of T is given by $|T|$. A database $\mathcal{DB}$ consists of a collection of relations $\{T_1, T_2, ..., T_l\}$.

A query q over the database $\mathcal{DB}$ is a conjunction of predicates, each of which indicates a constraint on an attribute (*e.g.*, A_2<3). The cardinality of a query q, $Card(q)$, is the number of tuples satisfying the query. Another related concept is selectivity, denoted by $Sel(q) = Card(q)/|T|$, *i.e.*, $Card(q)$ normalized by the table size (or the size of full outer join for join queries).

2.2 Problem Formulation

We formally introduce the problem:

Problem: Data Generation from Query Workloads. *Consider a set of n queries $Q = \{q_i\}_{i=1}^n$ and their cardinalities $\mathcal{K} = \{Card(q_i)\}_{i=1}^n$ collected on the database $\mathcal{DB}$, where n can be large. We aim to generate a database that satisfies the cardinalities of the collected queries (i.e., high fidelity to input cardinality constraints). We also expect the generated database to be close to the original database.*

This problem formulation follows the definition of *Data Generation Problem (DGP)* in previous work [4], which proves that this problem is NEXP-complete, and we can only design approximate algorithms. Moreover, we extend the target to recover the original database. This cannot be handled by [4], which will be explained later.

We use the cross entropy between the discrete data distributions of the generated relation $\widehat{T}$ and the original relation T to measure the similarity of two databases.

$$H(T, \widehat{T}) = -\mathbb{E}_{\mathbf{x}\sim T}[\log \widehat{Sel}(\mathbf{x})] \tag{1}$$

where $\widehat{Sel}(\cdot)$ denotes the selectivity of tuple $\mathbf{x}$ in the generated relation $\widehat{T}$.

Supported Queries and Join Schema. Our model supports learning from queries with conjunctions of predicates. We support range constraint, equality constraint or IN clause on a numerical or categorical column. Additionally, disjunctions can be supported using the inclusion-exclusion principle. For joins, we support foreign key joins. We also support multi-way and multi-key equi-joins. Following previous work on database generation and cardinality estimation [4, 14, 35], we also make a fairly natural assumption that all query constraints only have selection predicates on content columns (or value attributes), *i.e.*, no join key columns are filtered.

Moreover, unlike PGM-based methods [4] which merely support snowflake schema (tree structure with parent FK joining child PK), we only require the joins to be foreign key join with an acyclic schema (*i.e.*, a tree). We leave the support of cyclic joins to future work as the current autoregressive model architecture we use (which will be introduced later) only supports acyclic join schemas. Despite this, our method can support many practical workloads that are commonly used in large-scale cloud database scenarios as shown in Section 5.

Join Graph. We define the join graph of a join schema as follows. G_{join} is a tree-structured directed acyclic graph where the vertices $\mathcal{V}$ correspond to the relations $\mathcal{T}$ in the join schema. There exists a directed edge E from the vertex T_1 to the vertex T_2 if the primary key of relation T_1 joins with the foreign key of relation T_2.

2.3 Why not PGM

In this section, we briefly describe how PGM-based methods [4] for database generation work, and discuss the limitations of the methods, which our work aims to address.

How PGM-based Methods Work. PGM-based methods construct a Markov network $G = (\mathcal{V}, \mathcal{E})$ to represent the joint distribution of the attributes, where the vertices $\mathcal{V}$ correspond to the random variables $\mathcal{X}$ associated with each attribute, and an edge (X_i, X_j) exists if attributes A_i and A_j are filtered together in a cardinality constraint. It proposes algorithms based on Chordal graphs/Markov Blankets to factorize the joint distribution into marginal distributions, and solves a system of linear equations to obtain the marginal distributions. For multi-relation databases, it builds a separate PGM model for each view (including base relations) existing in the cardinality constraints, and then derives base relations from the generated views.

Limitation 1: Potentially Inaccurate Data Modeling. For tractability, PGM-based methods make independence assumptions between attributes. This may not hold for complex real-world datasets, which potentially leads to inaccurate data distribution modeling.

Limitation 2: Poor Scalability. The most computationally expensive part of PGM-based methods is solving the system of linear equations for the marginal distributions. The number of variables in the system is $O(lD^{\gamma})$, where l is the number of maximal cliques, and D is the domain size. Here, γ can reach the number of attributes in the worst case. D also scales linearly with regard to the number of queries n, until it reaches the actual domain size. In this case, the complexity of PGM is prohibitively expensive. This is confirmed in our experiments (Section 5) where it takes more than 12 hours to process 13 queries on the Census dataset. Therefore, PGM-based methods scale poorly with regard to the number of queries n, which may be large for real-world query workloads.

Limitation 3: Low Fidelity on Join Queries. The view generation procedure may result in violation of the containment property across generated views and relations because different views or relations consider disjoint sets of queries, *e.g.*, a value of relation T_2 in the generated join view (T_1, T_2) does not appear in the generated relation T_2. This would result in low fidelity of the generated database, *i.e.*, large error of input query cardinalities on the generated database.

3 OVERVIEW OF SAM

In this section, we first summarize several goals of the work and the corresponding solutions SAM uses to achieve the goals. Next, we present the high-level workflow of SAM.

3.1 Goals and Solutions

1. **Goal: Training an accurate generative model from cardinality constraints**. The first desiderata would be accurately modeling the data distribution from cardinality constraints. As explained earlier, the previous work [4] is not able to meet this goal since it makes independence assumptions on the data distribution. Moreover, the model must be generative so that we can efficiently sample tuples from the model.
   Solution. A recent advance in ML for cardinality estimation [34] proposes a unified deep autoregressive (AR) model — UAE, which models the data distribution from both data and query workloads without any independence assumptions. The query-driven variant of UAE, UAE-Q, is a deep AR model trained on cardinality constraints, which perfectly fits this goal. Note that this can address the Limitation 1 of PGM.
2. **Goal: Efficient processing of large-scale query workloads**. The previous work [4] can only process a small number of query constrains, but a real-world query workload may contain a large number of queries comprising multiple predicates with varying values on different attributes.
   Solution. This can be naturally achieved by our solution to Goal 1. This is because of 1) the batch training of deep AR models; 2) the acceleration of modern GPUs; 3) more importantly, the training complexity being *linear* in the number of collected queries, the number of attributes and the attribute domain size. Note that this can address the Limitation 2 of PGM.
3. **Goal: Effective and efficient generation of multiple base relations from an AR model**. With an AR model, we are able to uniformly sample full outer join tuples. However, uniform full outer join samples are biased for base relations due to the potential *fanout* effect, which is the phenomenon that join operations fan-out tuples from base relations, and a base relation tuple can appear multiple times in the full outer join result. This may cause the inconsistency between the marginal data distribution of base relation tuples in the full outer join and the original data distribution of the base relation tuples. Details can be found in Section 4.3. Therefore, we need to adjust the sampling process to produce unbiased samples for each base relation. Moreover, in a real-world database like IMDB, even the size of full outer join on a simple join schema containing 6 relations is $2 \cdot 10^{12}$, making it impractical to sample the entire set of full outer join tuples from the model. This requires us to design efficient sampling algorithms to generate the database.
   Solution. We use the *inverse probability weighting* [15, 24] technique to address the sampling bias. Specifically, for a full outer join sample, we adjust the weight of each base relation tuple according to the values of the fanout columns [14, 35]. However, it requires sampling the entire full outer join result to generate all base relations, which is infeasible as explained before. To overcome this, we sample a small fraction of full outer join result, and *scale* the sampled tuples to the actual size of the base relations. Our experimental results show that our solution can generate a database that maintains the data distributions of the original database by sampling only $1/20,000$ of the full outer join tuples.
4. **Goal: Assigning join keys to generated relations with high fidelity**. It depends on assigning the right join keys to the generated tuples of multiple base relations to maintain the data correlation across relations. A high fidelity database is generated only when both intra-relation data correlations and inter-relation data correlations are well captured.
   Solution. Based on the sampling mechanism of AR models, we design a novel *Group-and-Merge* algorithm for join key assignment. We identify sets of full outer join tuples that possibly share the same join key through *identifier column* values. We greedily merge tuple sets with the same *identifier columns* values and assign a unique key to them. Details can be found in Section 4.3.

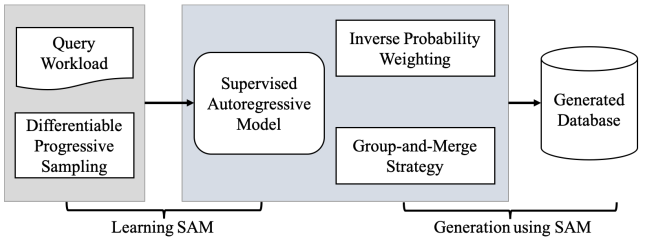
Query Workload
Differentiable Progressive Sampling
Supervised Autoregressive Model
Inverse Probability Weighting
Group-and-Merge Strategy
Generated Database
Learning SAM
Generation using SAM

Figure 1: Workflow of SAM.

Our approach to join key assignment helps the generated database to achieve 700× less error of input query cardinalities at tail compared to PGM-based methods [4]. Note that this can address the Limitation 3 of PGM.

3.2 Workflow of SAM

Figure 1 shows the high-level workflow of SAM. SAM consists of two stages — learning stage and generation stage. During the learning stage, batches of (query, cardinality) pairs (or query constraints) are collected from the log of the query workload. SAM learns the joint data distribution of the original database from the query constraints, using differentiable progressive sampling [34]. SAM can efficiently process a large-scale query workload, *i.e.*, 100K query constraints, within 48 hours.

During generation, SAM first samples full outer join tuples from the trained model, which can be done very efficiently on a GPU through batching. We then use *inverse probability weighting* to produce unbiased samples for each base relation, and assign join keys using the developed *Group-and-Merge* algorithm.

4 METHODOLOGIES

In this section, we first walk through the techniques used to build SAM — query-driven deep AR data modeling (Section 4.1). We then discuss how to use the constructed SAM to generate a database containing a single relation (Section 4.2), and a database containing multiple relations (Section 4.3).

4.1 Constructing SAM from Query Workload

Autoregressive Decomposition of Tabular Data Distribution. For a relation T and a set of attributes $A_1, ..., A_n$, the joint data distribution of T is given by:

$$P(x_1, ..., x_n) = f(x_1, ..., x_n)/|T| \tag{2}$$

where $f(\cdot)$ denotes the number of occurrences of a tuple. As it is hard to directly store the joint distribution $P(\cdot)$ due to the huge domain space $A_1 \times ... \times A_n$, factorization can be used to approximate the joint probability distribution with the product of a set of marginal and conditional probability distributions. While most factorization methods [13, 17, 31] make independence assumptions on the underlying data, the AR decomposition of the joint distribution factors $P(\cdot)$ without making any independence assumption. It thus considers the correlations among all attributes and is given by:

$$P(x_1, ..., x_n) = \prod_{i=1}^{n} P(x_i | x_1, ..., x_{i-1}). \tag{3}$$

Model Architectures. As shown in Figure 2, the input to the deep AR model of SAM is the range query $P_\theta(\wedge\{X_i \in R_i\})$, and the ultimate output is its corresponding cardinality estimate. Note that the intermediate output of the deep AR model is the conditional probability distributions $\{P_\theta(X_i|\mathbf{x}_{<i})\}$ for the i-th column given a data point $\mathbf{x}$, where θ is the model weights. To efficiently compute the ultimate cardinality result, we resort to sampling techniques to select a few data points to estimate. To this end, progressive sampling [36], which sequentially samples in-region values from the predicted distributions $\{P_\theta(X_i|\mathbf{x}_{<i})\}$ where $X_i \in R_i$, has shown to be robust to skewed data distribution. In addition, SAM can be instantiated by any learning-based AR architecture (*e.g.,* MADE [12] and Transformer [32]).

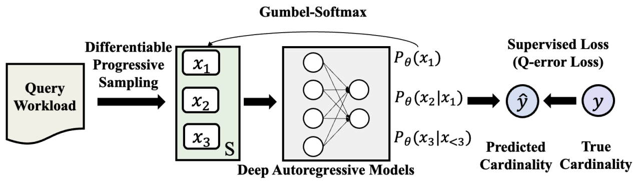

Figure 2: Constructing SAM from Query Workload. We temporarily use y to denote cardinality for simplicity.

Model Training. The original progressive sampling [36] method can only be used for query cardinality inference, but not for training from query cardinality constraints. This is because the categorically sampled variables $\mathbf{x}_i$ from $\{P_\theta(X_i|\mathbf{x}_{<i})\}$ are non-differentiable, which prevents the gradient flows during backpropagation. To enable training the deep AR model from query cardinality constraints, SAM uses *Differentiable Progressive Sampling* (DPS) [34], which applies the Gumbel-Softmax trick to Progressive Sampling. By making the sampled variables $\mathbf{x}_i$ differentiable with the Gumbel-Softmax trick, DPS enables gradients to flow through the categorically sampled variables, so that the AR model can be trained to minimize the discrepancy between the true and the estimated cardinalities. Figure 2 illustrates how DPS is done on a deep autoregressive model with three variables: the AR model first output $P_\theta(X_1)$, and $\mathbf{x}_1$ is sampled from $P_\theta(X_1 \in R_1)$ using Gumbel-Softmax. We then feed the sampled $\mathbf{x}_1$ into the AR model to predict the conditional probability $P_\theta(X_2|\mathbf{x}_1)$ and sample $\mathbf{x}_2$ from $P_\theta(X_2 \in R_2|\mathbf{x}_1)$. Similarly, we can obtain $P_\theta(X_3|\mathbf{x}_{<3})$ and sample $\mathbf{x}_3$. After obtaining all conditional probabilities, the predicted cardinality from the sample is given by $|T| \cdot \prod_{i=1}^{3} P_\theta(X_i \in R_i|\mathbf{x}_{<i})$. We can then calculate the Q-Error between the predicted cardinality and the actual cardinality, and perform backpropagation to calculate the gradient of the prediction error with regard to the model parameters, which can be used to update the model.

Join Handling. Following the previous work on deep AR models for tabular data modeling [35], SAM learns the correlations across all tables in a database using a *single* deep AR model of the full outer join. The reason for using full outer join instead of other join types is that full outer join encodes information of all tuples in the joint distribution, especially when some of the primary key relation tuples do not join with any foreign key relation tuple. SAM appends two types of virtual columns (*i.e.,* indicator and fanout columns) to the original content columns, and learns these columns altogether. One fanout column $\mathcal{F}_{T.\text{key}}$ is added for each foreign key T.key, which is defined as the number of times each value appears in T.key. For example, in Figure 3 (b), the fanout column $\mathcal{F}_{C.\text{x}}$ is 2 in the first tuple, because the foreign key $C.\text{x} = 1$ appears twice. One indicator column $\mathbb{I}_T$ is also added for each foreign key relation T, which takes on a binary value — 1 indicates relation T is present in the full outer join tuple, and 0 indicates otherwise. During query inference, SAM uses *fanout scaling* [35] in progressive sampling to eliminate potential fanout effects. We choose to build the single AR model for join handling as it is compatible with the DPS algorithm, and thus we can train the AR model from join queries as we do for single-relation queries.

4.2 Single Relation Generation

Once trained from the query workload, SAM is ready for database generation. We proceed to introduce the algorithm for generating a single-relation database using SAM.

For a single relation, SAM models the joint data distribution of all attributes in an autoregressive manner. Therefore, we can obtain uniform samples of the relation tuples by sequentially sampling the value for each attribute from the AR model. The target relation T is generated by sampling $|T|$ tuples.

Algorithm 1 presents the sampling algorithm for the case of a single table. To uniformly sample a tuple from the AR model, we first initialize an all-zero vector (line 3). We then sample the value of each attribute in a sequential way (line 4–7). For each attribute, we pass the previously sampled values to the AR model (line 5), sample the attribute value from the conditional probability distribution (line 6), and update the vector (line 7). By repeating the sampling procedure for $|T|$ times (line 2), we generate relation T.

Algorithm 1 Generating a Single Table with SAM

```
Input: Model predicted conditional probability distributions
    {$P_\theta(X_i|\mathbf{x}_{<i})$}; Table size $|T|$.
Output: Generated table $\overline{T}$.
1: $\overline{T} = \{\}$;              # Initialize $\overline{T}$ with an empty table
2: for $s = 1$ to $|T|$ do   # Embarrassingly parallel
3:     $\mathbf{z} = \mathbf{0}_n$;             # Initialize the $s$-th sample
4:     for $i = 1$ to $n$ do
5:         Forward pass and obtain $P_\theta(\mathbf{Z_i}|\mathbf{z}_{<i})$;
6:         Sample $z_i \sim P_\theta(\mathbf{Z_i}|\mathbf{z}_{<i})$;
7:         $\mathbf{z}[i] = z_i$;       # Assign the sampled $z_i$ to $\mathbf{z}$
8:     Append $\mathbf{z}$ to $\overline{T}$;
9: return $\overline{T}$
```

Note that the sampling process is *embarrassingly parallel*. Hence, in practice the sampling can be batched on a GPU and we can also launch multiple sampling threads on different GPUs, making the sampling process remarkably efficient.

4.3 Multiple Relation Generation

For a database containing multiple relations, we cannot generate the database by directly sampling from the AR model due to the following two challenges that need to be solved.

1. *Unbiased Sampling of base relation tuples.* In the case of multiple relations, SAM uses a single AR model to learn the joint distribution of the full outer join. *Direct sampling from the AR model only produces uniform samples of the full outer join*, rather than uniform samples of each base relation. Due to the potential fanout effect, the marginal data distribution of a base relation may no longer be maintained in the full outer join. As a result, we need a way to obtain unbiased samples of each base relation.
2. *Join key assignment.* For a database containing multiple relations, the relations contain join key columns, which specifies the data correlation across relations. However, join key columns are not modeled by the AR model, thus its value cannot be sampled. We need to assign join keys during the generation process in a way that maintains the inter-relation data correlations.

We proceed to present the algorithm to obtain unbiased samples of the base relations from the AR model (Section 4.3.1). We then discuss the algorithm to assign join keys, which maintains the inter-relation data correlations (Section 4.3.2).

4.3.1 *Unbiased Sampling of base relation tuples.*

Although SAM only models the joint distribution of the full outer join, we notice that the joint distribution of the full outer join (including fanout and indicator columns) carry information on the data distribution of each base relation. Specifically, each full outer join tuple contains the values for all base relation attributes, and the fanout column values provide information on the number of times each of the involved base relation tuple has been fanned out during the full outer join.

For a base relation, if we only consider the attributes of that base relation in the full outer join, then the full outer join is a pseudo-population of the base relation (target population) where the number of occurrences of each tuple is multiplied by the corresponding fanout factor. Because the full outer join is a pseudo-population for each base relation, uniform samples from the full outer join are biased samples for the base relations tuples. To illustrate this, we refer to Figure 3 (a). Both the tuple $(1, m)$ and $(2, m)$ appear once in the base relation A, and thus $P((1, m)) = P((2, m)) = 1/4$. However, in the full outer join of relation A, B and C shown in Figure 3 (b), there are two tuples containing $(1, m)$ and four tuples containing $(2, m)$ for relation A columns, *i.e.*, $P((1, m)) = 1/4, P((2, m)) = 1/2$, which is different from that in the base relation. This happens because tuple $(1, m)$ is fanned out twice when joining relation C, but tuple $(2, m)$ is fanned out twice when joining both B and C, leading to a total fanout of $2 \times 2 = 4$.

We propose to use *Inverse Probability Weighting* [15, 24], a well-known statistics technique, to remove this sampling bias. Inverse probability weighting is a general technique for calculating statistics standardized to a pseudo-population different from that in which the data was collected [29]. Given a full outer join sample $\mathbf{x}$, we derive a sample for each base relation, and we associate the derived sample for base relation T with a sample weight $\mathcal{W}_T(\mathbf{x})$.

$$\mathcal{W}_T(\mathbf{x}) = \frac{1}{\prod_{T' \notin \{T\} \cup Ancestor(T)} \mathcal{F}_{T'.\text{key}}} \tag{4}$$

$\mathcal{W}_T(\mathbf{x})$ is the inverse of the fanout factor of the base relation T tuple, *i.e.*, the number of times the tuple is fanned out in the full outer join. Fanout factor of a base relation T tuple can be calculated by multiplying the fanout column value of all relations, excluding relation T and its ancestors in the join graph.

Example. The table in Figure 3 (c) shows 4 samples from SAM. Without loss of generality, we focus on relation A in the second tuple, and we have $\mathcal{W}_A = \frac{1}{\mathcal{F}_{B.x} \times \mathcal{F}_{C.x}}$. For this tuple, $\mathcal{F}_{B.x} = \mathcal{F}_{C.x} =$

$$\begin{tikzcd} B.x & A.x \arrow[l] \arrow[r] & C.x \end{tikzcd}$$

A.x	A.a
1	m
2	m
3	n
4	n

B.x	B.b
1	a
2	b
2	c

C.x	C.c
1	i
1	i
2	j
2	j

(a) Base relations and join graph

$A.x$	$A.a$	$\mathcal{F}_{B.x}$	$B.b$	$\mathcal{F}_{C.x}$	$C.c$
1	m	1	a	2	i
1	m	1	a	2	i
2	m	2	b	2	j
2	m	2	b	2	j
2	m	2	c	2	j
2	m	2	c	2	j
3	n	1	∅	1	∅
4	n	1	∅	1	∅

(b) Full outer join.

Tuple	1	2	3	4
$\mathcal{W}_A^s$	1	0.5	0.5	2

$A.a$	$\mathcal{F}_{B.x}$	$B.b$	$\mathcal{F}_{C.x}$	$C.c$	$\mathbb{I}_B$	$\mathbb{I}_C$	$\mathcal{W}_A$	$\mathcal{W}_B$	$\mathcal{W}_C$
m	1	a	2	i	1	1	0.5	0.5	1
m	2	b	2	j	1	1	0.25	0.5	0.5
m	2	c	2	j	1	1	0.25	0.5	0.5
n	1	∅	1	∅	0	0	1	∅	∅

(c) Samples from SAM and scaling.

		$A.a$	$\mathcal{F}_{B.x}$	$B.b$	$\mathcal{F}_{C.x}$	$C.c$	$\mathcal{W}_A^s$	$\mathcal{W}_B^s$	$\mathcal{W}_C^s$
	Group 1	m	1	a	2	i	1	1	2
Merge	Group 2	m	2	b	2	j	0.5	1	1
	Group 2	m	2	c	2	j	0.5	1	1
	Group 3	n	1	∅	1	∅	2	∅	∅

(d) Group-and-Merge Algorithm.

A.x	A.a
1	m

B.x	B.b
1	a

C.x	C.c
1	i
1	i

(e) Group 1

A.x	A.a
2	m

B.x	B.b
2	b
2	c

C.x	C.c
2	j
2	j

(f) Group 2

A.x	A.a
3	n
4	n

(g) Group 3

Figure 3: Example.

2, which indicates that the original tuple of base relation A joined in the full outer join is fanned out twice by both relation B and relation C. Therefore, the sample weight $\mathcal{W}_A = \frac{1}{2\times 2} = 0.25$.

Theorem 1. *Sampling from full outer join with weight $\mathcal{W}_T$ for relation T is asymptotically unbiased.*

Proof: To prove Theorem 1, we need to show that the sample distribution asymptotically approximates the true probability distribution, *i.e.*, the expected frequency of any relation T tuple $\mathbf{x}$ in the sample approaches its occurring probability $P(\mathbf{x})$ as the sample size approaches infinity. Inspired by [15], we first calculate the inclusion probability $\pi_{\mathbf{x}}$ of relation T tuple $\mathbf{x}$ in the sample. We denote the full outer join size by N, the size of relation T by N_T, and the sample size by S. We denote the number of occurrences of tuple $\mathbf{x}$ in relation T and in the sample by $N_T(\mathbf{x})$ and $S(\mathbf{x})$, respectively.

$$\pi_{\mathbf{x}} = \frac{1}{N_T(\mathbf{x})} \cdot S \cdot \frac{N_T(\mathbf{x})/\mathcal{W}_T(\mathbf{x})}{N} = \frac{S}{\mathcal{W}_T(\mathbf{x}) \cdot N} \tag{5}$$

We then prove that the expected frequency $\mathbb{E}[\widehat{P}(\mathbf{x})]$ approaches its occurring probability $P(\mathbf{x})$ as $S \to \infty$. Note that $\sum_{i=1}^{S} W_T(\mathbf{x}_i) \to \frac{S\cdot N_T}{N}$ as $S \to \infty$.

$$\begin{aligned}
\mathbb{E}[\widehat{P}(\mathbf{x})] &= \mathbb{E}[\frac{S(\mathbf{x}) \cdot W_T(\mathbf{x})}{\sum_{i=1}^{S} W_T(\mathbf{x}_i)}] \\
&= \mathbb{E}[\sum_{i=1}^{S} \frac{\mathbb{1}_{\mathbf{x}_i=\mathbf{x}} \cdot \mathcal{W}_T(\mathbf{x}_i)}{\sum_{i=1}^{S} W_T(\mathbf{x}_i)}] \\
&= \mathbb{E}[\sum_{i=1}^{N_T} \mathbb{1}_{\mathbf{x}_i \in S} \cdot \frac{\mathbb{1}_{\mathbf{x}_i=\mathbf{x}} \cdot \mathcal{W}_T(\mathbf{x}_i) \cdot N}{S \cdot N_T}] \\
&= \frac{1}{N_T} \cdot \sum_{i=1}^{N_T} \pi_{\mathbf{x}_i} \cdot \frac{\mathbb{1}_{\mathbf{x}_i=\mathbf{x}}}{\pi_{\mathbf{x}_i}} = P(\mathbf{x})
\end{aligned} \tag{6}$$

Handling NULL value in full outer join. NULL value may appear in the full outer join when some of the primary key relation tuples do not join with any foreign key relation tuple. Given a full outer join sample with NULL values, we only derive samples for base relations that are not NULL. Base relations that are not NULL are indicated by a corresponding indicator column value of 1, *i.e.*, those relations actually participate in the join. For base relations with NULL values, the associated fanout column values are set to 1 during weight calculation. For example, for the fourth sample in Figure 3 (c), the indicator columns of both relation B and C are 0, so we can only derive relation A sample from it, and the weight $\mathcal{W}_A$ is calculated by setting both fanout columns $\mathcal{F}_{B.x}$, $\mathcal{F}_{C.x}$ to 1.
Challenge: large join space. To generate a set of tuples equal to the size of the base relations using *inverse probability weighting*, the sample size needs to equal that of the full outer join. However, the full outer join is usually prohibitively large for real-world databases. For example, the size of full outer join on a simple join schema (6 tables) of the IMDB database can reach $2 \cdot 10^{12}$. Consequently, it is only feasible to sample a small fraction of the full outer join.

To avoid generating the full outer join and ensure that base relations in the generated database are of their actual sizes, we adopt a simple and effective *scaling* method for each base relation. Specifically, for a base relation T, we look up and accumulate the values of $\mathcal{W}_T$ of samples from SAM and obtain the cumulative sum $\mathcal{W}_T^{\text{sum}}$. Then, we multiply each sample weight $\mathcal{W}_T$ with a scaling factor of $|T|/\mathcal{W}_T^{\text{sum}}$. $\mathcal{W}_T^{\text{sum}}$ implies the number of tuples the full outer join samples can generate for relation T. Therefore, by multiplying the sample weights with the scaling factor $|T|/\mathcal{W}_T^{\text{sum}}$, we generate exactly $|T|$ tuples for relation T.

Example. Again we focus on the below table in Figure 3 (c). For relation A, $\mathcal{W}_A^{\text{sum}} = 0.5+2\times 0.25+1 = 2$ and $|A|$ is 4. The corresponding scaling factor is thus 2. The above table in Figure 3 (c) shows the sample weight of relation A after scaling, $\mathcal{W}^s_A$. The cumulative sum of $\mathcal{W}^s_A$ is 4, the same as $|A|$.

After this, we can generate the tuples for each base relation T using $\mathcal{W}^s_T$ — the number of occurrences of each tuple should be equal to the corresponding $\mathcal{W}^s_T$ value.

The remaining issue here is that the generated tuples do not contain join keys, because join key columns are not explicitly modeled by SAM. We discuss the algorithms to generate base relations with join keys in Section 4.3.2. For now, we present the algorithm to generate multiple base relations (without join key) in Alg 2. The cumulative sum of each base relation's weight is initialized to 0 (line 2). We first uniformly sample full outer join tuples from the AR model using the same sampling technique as Alg 1 (line 4). Then, for each base relation, we use *inverse probability weighting* to calculate the sample weight (line 6), and append the weighted sample to the set of generated tuples (line 7). The calculated weight is also added to the corresponding cumulative sum (line 8). After processing all k full outer join samples, we calculate the scale factor for each base relation (line 10), and adjust the number of tuples for each base relation according to the scale factor (line 11).

Algorithm 2 Generating Multiple Tables with SAM (no join key)

```
Input: Model predicted conditional probability distributions
    $\{P_\theta(X_i|\mathbf{x}_{<i})\}$, sample size $k$
Output: Generated tables $\overline{T}_1, \overline{T}_2, ...., \overline{T}_l$
1:  $\overline{T}_1, \overline{T}_2, ...\overline{T}_l = \{\}$;
2:  $\mathcal{W}^{sum}_{T_1}, \mathcal{W}^{sum}_{T_2}, ..., \mathcal{W}^{sum}_{T_l} = 0$
3:  for $s = 1$ to $k$ do
4:      Uniformly sample full outer join tuple $\mathbf{z}$ (line 3-7 of Alg 1)
5:      for $i = 1$ to $l$ do  # Inverse Probability Weighting
6:          $\mathcal{W}_{T_i} = \frac{1}{\prod_{T' \in Fanout(T_i)} \mathcal{F}_{T'.\text{key}}}$
7:          Append $(\mathbf{z}[T_i], \mathcal{W}_{T_i})$ to $\overline{T}_i$
8:          $\mathcal{W}^{sum}_{T_i} = \mathcal{W}^{sum}_{T_i} + \mathcal{W}_{T_i}$
9:  for $i = 1$ to $l$ do    # Scaling
10:     $scale\_factor = |T_i| / \mathcal{W}^{sum}_{T_i}$
11:     Scale number of tuples in $\overline{T}_i$ by $scale\_factor$
12: return $\overline{T}_1, \overline{T}_2, ...., \overline{T}_l$
```

4.3.2 *Join key Assignment.*

Join key assignment is crucial in order for the generated base relations to capture the inter-relation data correlation of the original database. If we randomly assign join keys to the base relations generated by Alg 2, they are unlikely to recover the full outer join distribution modeled by SAM when joined together.

In order to achieve high-fidelity on join queries, we need to assign the right join keys to the samples during the generation process, such that the generated base relations, when joined together, produce full outer join result that follows the distribution modeled by SAM. In doing so, the error of input query cardinalities on the generated database could be minimized as well.

We could always assign values to the primary key columns sequentially, and thus we are left with assigning values to the foreign key columns. Since we can use SAM to generate an arbitrary view (containing a subset of the base relations) with the *inverse probability weighting* approach by considering the fanout columns associated with the target view, one naive approach to join key assignment following [4] would be to first generate the relation containing the primary key (*a.k.a.,* pk relation) and a view containing the pk relation and target relation (*a.k.a.,* fk relation). Then we could derive the foreign key for the target relation from the pk relation and the view.

As shown in the example below, while this approach can capture the data correlation between pairs of pk relation and fk relation, it does not maintain the joint distribution of full outer join, *i.e.*, the data distribution of all attributes for each relation as well as the data correlation across different relations.

Example. Following the example in Figure 3, assume we generate relation A (Figure 4 (a)), and two views (A, B) and (A, C) (Figure 4 (b)). To derive the foreign key of relation B and C, for each tuple in the views, we look up tuples in the relation A where A.a matches, and assign the primary key value to the respective foreign key. However, problems occur when processing the column (m, a) in the view of (A, B) — both tuples in relation A $(1, m)$ and $(2, m)$ matches. As we do not have any clue beyond the two relation views, we can at best randomly select a primary key value from all matched tuples in relation A. As shown in Figure 4 (c), this may end up with assigning 1 to the foreign key of all relation B tuples, and assigning 2 to the foreign key of all relation C tuples, leading to an empty inner join result. Consequently, assigning join key from views breaks inter-relation data correlation between relation B and C.

$A.x$	$A.a$
1	m
2	m
3	n
4	n

(a) PK relation

$A.a$	$B.b$
m	a
m	b
m	c

$A.a$	$C.c$
m	i
m	i
m	j
m	j

(b) 2-relation views

$B.x$	$B.b$
1	a
1	b
1	c

$C.x$	$C.c$
2	i
2	i
2	j
2	j

(c) FK relations

Figure 4: Example of assigning join keys from views.

The key reason why assigning join key from the join views as above fails is that the views only maintain the marginal joint distribution of the involved relations, rather than the distribution of the full outer join. As a result, the join keys assigned independently from each view fail to capture the joint data distribution of the entire database.

A high-level idea to address this is to derive the join relationships directly from the full outer join and assign join key(s) accordingly, so that the data correlation across *all* columns in the database can be maintained. Fortunately, uniform samples from the full outer join can be easily obtained from the AR model in SAM, which is not the case for PGM-based methods since the model of each view (including full outer join view) only uses partial information contained in the input query constraints, as discussed in Section 2.3. Therefore, given samples from the full outer join, our target is then to assign join key(s) to the samples, such that when base relations are generated from them, the full outer join of the generated base relations could match that of the full outer join samples.

Thus, we need to identify the full outer join tuples that share a join key, and assign them with a unique join key. The base relation tuples are then generated from the full outer join tuples, which already have join keys assigned, and take on the same join key of the full outer join tuples. The crux of the problem is then to identify full outer join tuples sharing the same join key. The following theorem gives hints about this.

THEOREM 2. *Assuming an AR model of the exact joint distribution, full outer join tuples sharing the same join key $T.pk$ have the same value for all indicator columns and content columns associated with $\{T\} \cup Ancestor(T)$ in the join graph, as well as for all fanout columns associated with fk relations that join with $\{T\} \cup Ancestor(T)$.*

We can prove Theorem 2 based on the uniqueness of primary key. We denote the set of all indicator columns and content columns associated with $\{T\} \cup Ancestor(T)$ and all fanout columns associated with fk relations that join with $\{T\} \cup Ancestor(T)$ as the *identifier columns* of the join key $T.pk$, or $Identifier(T.pk)$.

Example. In Figure 3 (b), relation A's primary key $A.x$ joins with the foreign key $B.x$ and $C.x$. As A has no ancestor on the join graph, the *identifier columns* of relation A are all indicator columns and content columns associated with relation A, and all fanout columns associated with relation B and C, *i.e.*, $Identifier(A.x) = \{A.a, \mathbb{I}_A, \mathcal{F}_{B.x}, \mathcal{F}_{C.x}\}$. Full outer join tuples 3 to 6 share the join key 2, and consequently they have the same values across all *identifier columns* of $A.x$.

Theorem 2 states a necessary condition for full outer join tuples to have the same join key. Based on this condition, we can formulate a greedy algorithm for join key assignment, where we identify tuples satisfying the necessary condition, greedily merge them together and assign them with a unique join key. We call this algorithm *Group-and-Merge*, as it consists of two steps. Specifically, to assign the join key $T.pk$, we first group the full outer join tuples by the *identifier columns* of $T.pk$. At the second step, as the scaled weight of pk table $\mathcal{W}^s_T$ for some tuples might be less than 1, we greedily merge tuples within the same group together, keep track of the cumulative sum of the scaled weight $\mathcal{W}^s_T$, and assign the merged tuples with a unique key whenever the scaled weights sum to 1, *i.e.*, a new tuple for the primary key relation T is generated.

The *Group-and-Merge* algorithm is presented in Alg 3. Initially *current_sets* stores all the full outer join samples with *inverse probability weighting* and *scaling* applied (line 2). We use *merged_set* to temporarily store the merged tuple sets during the merging process (line 3). We also maintain a counter for the pk relation T to record the number of primary keys assigned (line 4). To assign the join key $T.pk$, we *group* the tuples by values of the *identifier columns* of $T.pk$ (line 5). From our insight, we know that tuples grouped together likely share the same $T.pk$, so we greedily *merge* tuples within each group (line 6-17). Specifically, we use *set_to_merge* to store tuple sets that are to be merged (line 7), and *weight_sum* to track the cumulative sum of the scaled weight $\mathcal{W}^s_T$ of the merged tuples (line 8). With each tuple added to *set_to_merge*, we increment the weight in *weight_sum* (line 10-11). We assign the merged tuples in *set_to_merge* with a unique key when *weight_sum* reaches 1 (line 12-13), *i.e.*, a new tuple has been generated for the pk relation T, and they are added to *merged_sets* as a merged tuple set (line 14). The counter value is also incremented by 1 to record the assigned key (line 15). *set_to_merge* and *weight_sum* are reset after assigning a new key (line 16-17). After all samples have join keys assigned, each base relation T_i can be generated from them according to the scaled weight $\mathcal{W}^s_{T_i}$ of the samples (line 18). Note that Alg. 3 can be easily extended to handle multiple join keys by merging samples in a recursive manner. Due to the page limit, we present the case of multiple join keys in the full version.

Algorithm 3 Generating Multiple Tables with SAM (join key assigned using *Group-and-Merge*)

```
Input: Full outer join samples X after applying inverse probability
    weighting and scaling.
Output: Base relation tuples with join key $T.pk$ assigned.
1: $\mathcal{DB} = \{\}$
2: $current\_sets = \{X_1, X_2, ...X_n\}$
3: $merged\_sets = \{\}$
4: $counter = 0$
5: Group $current\_sets$ by values of identifier columns
    $Identifier(T.pk)$ into $\{\mathbf{X}_{1_1}, \mathbf{X}_{1_2}...\}$, $\{\mathbf{X}_{2_1}, \mathbf{X}_{2_2}...\}$, ... ,
    $\{\mathbf{X}_{k_1}, \mathbf{X}_{k_2}...\}$
6: for each group $\{\mathbf{X}_{k_1}, \mathbf{X}_{k_2}, ...\}$ do
7:     $set\_to\_merge = \{\}$
8:     $weight\_sum = 0$
9:     for each sample $\mathbf{X}_{k_i}$ do
10:        Append $\mathbf{X}_{k_i}$ to $set\_to\_merge$
11:        $weight\_sum = weight\_sum + \mathcal{W}^s_T$
12:        if $weight\_sum >= 1$ then
13:            Assign key $counter$ for the primary key column $T.pk$
               to all tuples in $set\_to\_merge$
14:            Append $set\_to\_merge$ to $merged\_sets$
15:            $counter = counter + 1$
16:            $set\_to\_merge = \{\}$
17:            $weight\_sum = 0$
18: Generate database $\mathcal{DB}$ from $merged\_sets$ according to the
    scaled weight $\mathcal{W}^s_{T_i}$ of the samples
19: return The generated database $\mathcal{DB}$
```

Example. Continue with our example, after applying inverse probability weighting and scaling to the samples, we obtain each tuple's scaled weight for all base relations as shown in Figure 3 (d). Now we can apply the Group-and-Merge algorithm to generate the base relations while assigning join keys. We keep a counter for relation A, and the counter value is initialized to 0. We group the tuples by the values of the *identifier columns* of $A.x$, which are $\{A.a, \mathbb{I}_A, \mathcal{F}_{B.x}, \mathcal{F}_{C.x}\}$. Three groups are formed, the first tuple belongs to Group 1, the second and third tuple belong to Group 2, and the fourth tuple belongs to Group 3. We start by processing Group 1. We add $\mathcal{W}^s_A = 1$ to *weight_sum* and it reaches 1, indicating a new tuple is generated for relation A. We assign a new primary key 1 to the tuple, and increment the counter value by 1. Base relation tuples generate from this sample are shown in Figure 3 (e), which include one tuple for relation A, one tuple for relation B and two tuples for relation C. All the tuples generated share the join key 1. Moving on to Group 2, we first process the second tuple, append it to *set_to_merge*, and add $\mathcal{W}^s_A = 0.5$ to *weight_sum*, which becomes $0 + 0.5 = 0.5$. Then we process the third tuple, append the tuple *set_to_merge*, and add $\mathcal{W}^s_A = 0.5$ to *weight_sum* to become

$0.5 + 0.5 = 1$, indicating another new tuple is generated for relation A. Thus the second and third tuples are merged, and we assign a new primary key 2 to the merged set. We can then generate tuples for all base relations from the merged set by aggregating the scaled weights, as shown in Figure 3 (f). All the tuples generated share the join key 2. For Group 3, we repeat the exact steps to generate 2 tuples for relation A as show in Figure 3 (g). Note that no tuples are generated for relation B and C from Group 3 because the corresponding indicator column values of both relations are 0. Now all the generated tuples together form the generated database, and it is exactly the same as the original database.

Handling numerical columns. Numerical columns with large domain sizes may pose a challenge to SAM. Due to the approximation error of the deep autoregressive model, we may obtain samples that are supposed to share a join key, but have different numerical values. This prevents them from being identified and merged together by the *Group-and-Merge* algorithm, leading to decreased data fidelity. To tackle this problem, we use the intervalization technique [4] and integrate this into the deep autoregressive model. The intervalization technique models the distribution of intervals, *i.e.*, value ranges, instead of distinct values for numerical columns, which greatly reduces the domain size of numerical columns. To apply the intervalization technique, we extract all distinct constants that appears in the predicates on a numerical column from the training queries, and sort them in increasing order $v_1, v_2, ..., v_l$. We then define $l - 1$ intervals $[v_i, v_{i+1})(1 \leq i < l)$, and train SAM to learn the conditional probability of the column's value falling in each interval. As there are far fewer intervals compared to distinct values, this reduces the domain size of numerical columns. In the generation phase, we sample the interval for each numerical column from the autoregressive model, and perform *Group-and-Merge* based on the intervals. After *Group-and-Merge*, for each generated tuple containing numerical columns, we perform uniform random sampling from distinct values in the interval to obtain the actual value for the numerical columns.

5 EXPERIMENTS

We conduct comprehensive experiments to evaluate SAM [1] in comparison with previous methods. We evaluate the accuracy of database generation methods in two aspects.

A1. *Fidelity* of the generated database to input cardinality constraints.

A2. *Closeness* of the generated database to the original database. We term this aspect as *database recovery*.

Moreover, we also evaluate the efficiency of database generation methods in the two aspects.

E1. Efficiency of the *processing* of input cardinality constraints.

E2. Efficiency of the database *generation* process.

5.1 Experimental Settings

Datasets. We use three real-world datasets, including two single-relation datasets and one multi-relation dataset, in our experiments. A brief overview of the datasets is as follows.

1. **DMV** [37]. This dataset consists of vehicle registration information in New York. We follow the preprocessing strategy in previous work [34, 36], and get 11.6M tuples and 11 columns after preprocessing. The 11 columns have widely different data types and domain sizes ranging from 2 to 2101.
2. **Census** [2]. This dataset was extracted from the 1994 Census database, consisting of personal income information. It contains 48K tuples and 14 columns. The 14 columns contain a mix of categorical columns and numerical columns with domain sizes ranging from 2 to 123.
3. **IMDB** [18]. The Internet Movie Database (IMDB) is a public database containing information related to films and television programs. IMDB has been proved to contain relatively strong data correlation [18] and is widely used in database research on complex real-world data. We use the JOB-light [19] schema in our experiments, which is a widely-used benchmark on IMDB and has 6 joined relations. The size of full outer join is $\sim 2 \cdot 10^{12}$.

Query Workload. For the two single-relation databases, as there is no available real query workload on these datasets, we generate query workloads following previous work on cardinality estimation [7, 16]. Specifically, we draw the number of filters (n_f) from 1 to 5 at random. We then uniformly sample n_f columns and the respective filter operators from $\{\leq, =, \geq\}$. Finally, we assign the filter literals by the values of a tuple uniformly sampled from the datasets. We generate a query workload of 20K queries for each database, which is orders of magnitude larger than the setting of previous work [4]. We also generate another query workload containing 100K queries to further show the ability of SAM to efficiently process larger-scale query workloads, and evaluate the trend of **A2** (Closeness) of SAM with varying number of input queries. For the multi-relation IMDB database, we directly use the training queries generated by previous work on cardinality estimation [16] as our input query workload. This query workload consists of 100k queries and the number of joins ranges from 0 to 2. The number of filters for each base relation is randomly drawn from 0 to the number of columns. There are two reasons for the choice. First, it is a large-scale query workload comprising a wide variety of both single-relation and join queries, which mimics real-world query workloads. Second, it has been released to the public.

Metrics. Based on the two aspects of accuracy mentioned (**A1.** Fidelity and **A2.** Closeness), we use a number of metrics to measure the performance of different database generation methods.

Aspect 1: Evaluation on fidelity. Our first experiment evaluates the fidelity of the generated database, *i.e.*, how well the generated database satisfies the input cardinality constraints. This can be measured by the relative cardinality errors of input queries on the original and the generated databases. To this objective, we use Q-Error [25] of input query cardinalities as our metric, which is a popular metric for cardinality estimation [16, 34, 36]. Note that due to the scale of the IMDB training query workload (100K), it is impractical to evaluate the generated database on all input query constraints. Therefore, we evaluate the generated database on a random sample of $1,000$ input query constraints for IMDB.

Aspect 2: Evaluation on Closeness to the original database. Second, we evaluate how close the generated database is to the original database in terms of data distribution. Two metrics are used in this experiment: 1) cross entropy with regard to the original database; 2) Q-Error on an independent test query set.

[1]The source code will be released.

As discussed in Section 2.2, cross entropy directly measures the closeness between the data distribution of the generated database and the original database. We calculate the cross entropy in bits between the relation(s) in the original database and the relation(s) in the generated database. Smaller cross entropy indicates that the generated database is statistically closer to the original database. For IMDB database, we measure the cross entropy of the primary key relation — Title.

Apart from measuring the cross entropy, we can measure the closeness to the original database by measuring the Q-Error on an independent test query set. Unlike the Q-Error on the input query workload, which measures how well the input query cardinalities are satisfied, Q-Error on an independent test query test measures how well the generated database recovers the joint data distribution and generalizes to unseen queries.

As our method serves the use cases of benchmarking and stress testing, we also evaluate how the query execution latency differs between the generated database and the original database. Therefore, following [21] we measure the performance deviation of an independent test workload for each synthetic database. Performance deviation is defined as the difference in query latency between the synthetic database and the original database. For all three datasets, we evaluate the query latency of an independent test workload on the original database and the synthetic databases using the open source DBMS PostgreSQL 12.0. We then calculate the performance deviation for each synthetic database.

On single relation databases, the test query workloads are randomly generated with the same procedure as the input query workloads but are ensured to have no duplicate query. For IMDB, we use the 70 queries in the JOB-light benchmark as our independent test query set. While MSCN training queries only contain joins of up to three relations, JOB-light contains a variety of join queries involving up to five relations. Therefore, JOB-light queries can especially help evaluate how well the joint distribution of all relations are captured by different methods.

Baseline Methods. We implement the PGM-based method in previous work [4], specifically the chordal graph-based method, as the baseline method. As discussed, PGM [4] is the only comparable method for database generation that shares the same problem settings with our work.

Evaluation Protocols. We feed a query workload into different methods. After processing the query workload, we generate the database using these methods. We then execute the input queries on the generated database and measure the metrics of different aspects discussed above. All the experiments were run on a machine with a Tesla V100 GPU and a 20-core E5-2698 v4 @ 2.20GHz CPU.

Notably, as PGM has a high time complexity with regard to the size of the query workload, it can only process a very small number of query constraints within a reasonable time frame, which we discuss in detail in Section 5.2. Considering the huge discrepancy in processing efficiency between different methods, we fix the processing time instead of input query size for our evaluation on database recovery (Section 5.4), *i.e.*, we use different methods to process as many queries as possible within a fixed time frame (12 hours for single relation databases and 48 hours for IMDB). Under the fixed time frame, PGM can process 12, 7 and 400 queries for Census, DMV and IMDB, respectively.

To fairly compare SAM's performance of generated data fidelity with the previous method (Section 5.3), we need to evaluate the methods on the *same* set of input query constraints. In this case, we evaluate the performance of both SAM and PGM on the small input query workload which PGM can fully process within the fixed time frame. The results are shown in Table 2 and Table 4. However, we note that such a small size of query workload can hardly help recover the database, and real-world query workloads usually contains a much larger number of queries. Therefore, results in Table 2 and Table 4 are presented just for fairness of comparison.

5.2 How Fast is SAM in processing Query Workloads?

To evaluate the efficiency of compared methods, we first study the query workload processing time. Processing time refers to the time it takes for a method to model the data distribution from the given input cardinality constraints.

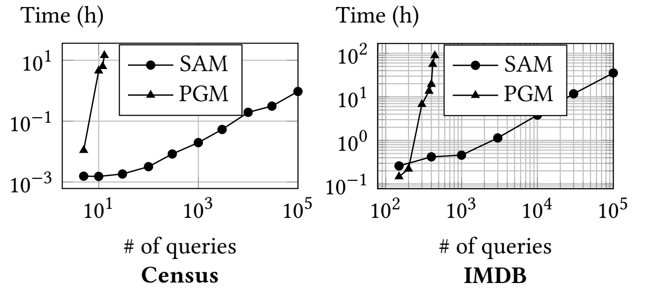

Figure 5: Processing time.

Figure 5 shows the log-log plot of the processing time of SAM and PGM against the number of input queries on the single-relation database Census and the multi-relation database IMDB, respectively. Note that since the result on DMV demonstrates the same trend as that on Census, we do not report it due to the space limit. We observe that the processing time scales as a high-degree *polynomial* for PGM and *linearly* for SAM. We observe that PGM can only process very few cardinality constraints within a reasonable time frame, as it takes more than 12 hours to process 13 queries on Census and more than 48 hours to process 410 queries on IMDB[2], and a small increase in the query workload size leads to a huge increase in the processing time. On the contrary, SAM can process the full set of input cardinality constraints well within 12 hours on Census (20K) and 48 hours on IMDB (100K). Moreover, note that the average number of filters of the input queries on the two single-relation databases is 2.5. If we increase the average number of filters, PGM would become even more inefficient. **Therefore, SAM is orders of magnitude more efficient than the baseline method in processing query workloads, and only SAM can scale to large-scale query workloads**.

5.3 How do the Methods Compare in Fidelity?

Table 1~Table 2 and Table 3~Table 4 show the experimental results of different data generation methods in terms of Fidelity on single- and multi-relation databases, respectively. We first highlight the

[2]Since input queries are assigned to different PGM models according to the joined tables on IMDB, the number of processed queries on IMDB is larger than that on Census, but still account for a very small fraction of the input queries.

Model	Census				DMV			
	Median	75th	90th	Mean	Median	75th	90th	Mean
SAM	1.27	1.65	2.50	1.80	1.15	1.48	2.28	2.10

Table 1: Q-Error of input queries - full scale

Model	Census (12 queries)				DMV (7 queries)			
	Median	75th	90th	Mean	Median	75th	90th	Mean
PGM	1.05	1.65	6.99	2.61	1.00	1.04	1.06	1.02
SAM	1.32	1.56	1.63	1.84	2.81	8.41	15.69	5.97

Table 2: Q-Error of very few input queries

Model	Median	75th	90th	Mean	Max
SAM w/o Group-and-Merge	2.00	4.68	26.00	2602	$2 \cdot 10^6$
SAM	1.57	2.61	5.74	14.85	3142

Table 3: Q-Error of input queries on IMDB - full scale

Model	Median	75th	90th	Mean	Max
PGM	1.55	149.5	6202	$1 \cdot 10^5$	$1 \cdot 10^7$
SAM w/o Group-and-Merge	1.98	5.24	24.34	$2 \cdot 10^4$	$4 \cdot 10^6$
SAM	1.77	3.58	8.60	17.97	5040

Table 4: Q-Error of 400 input queries on IMDB

main takeaway from the results — **SAM enjoys a significantly better trade-off between accuracy and scalability compared to the baseline method.** Below we analyze the detailed experimental results and conclude several major findings from the results.

(F1) SAM achieves great performance in terms of data fidelity on large query workloads on all databases. Table 1 and Table 3 show the Q-Error of input query cardinality on large query workloads on single- and multi-relation database, respectively. Note that PGM cannot work on such workloads. From the results, we observe that SAM performs extremely well on single-relation databases, achieves single-digit error across all percentiles on Census and DMV dataset. SAM performs well on IMDB dataset too, which contains join queries in the query workload. With the *Group-and-Merge* algorithm for join key assignment, SAM's performance improves significantly at the tail, achieving 500× less Max error compared to the model without *Group-and-Merge*. SAM's performance is mainly attributed to its accurate modeling of the joint data distribution without making compromising independence assumptions, as well as the properly assigned join keys.

(F2) SAM has comparable performance with PGM in terms of data fidelity on very small query workloads on single-relation databases . As PGM can complete in reasonable time only if the workload is very small, to compare with PGM, we use workload with several queries only. As shown in Table 2, on a small query workload, both SAM and PGM are able to achieve relatively small Q-Error of input query cardinality. In other words, both methods generate a single-relation database well satisfying the input cardinality constraints. SAM outperforms PGM in terms of 75th, 90th and Mean error on the Census dataset, while PGM outperforms SAM on the DMV dataset. This is because given a very small number of cardinality constraints, PGM derives a near-exact solution by solving the system of linear equations, achieving high fidelity. Meanwhile, SAM uses the AR model to learn an approximate solution. The approximate solution can incur more error compared to the solution of PGM, but most of the time its performance is on par with that of PGM.

(F3) SAM significantly outperforms PGM on small query workload on multi-relation databases. From Table 4, we observe that SAM outperforms PGM for all percentiles except median when measuring the Q-Error of the small query workload for the IMDB dataset, which contains join queries. SAM performs particularly well for the larger percentiles, *i.e.*, 75th, 90th and Max, achieving 40×, 700×, 2,000× less error respectively. As SAM uses an AR model to learn the joint data distribution of the full outer join from all input query constraints, and uses the *Group-and-Merge* algorithm to ensure generated base relations recover the full outer join, it leads to a generated database that recover the joint distribution of the full outer join to the greatest extent. On the contrary, PGM independently generates different views satisfying disjoint sets of input query constraints, leading to inconsistencies across the generated views and huge error at the tail percentiles. PGM performs slightly better than SAM in terms of the median, as it obtains a near-exact solution for a portion of the queries, but this is at the cost of incurring much larger error for the rest of the queries.

5.4 How do the Methods Compare in Database Recovery?

Table 5~Table 9 show the experimental results of different data generation methods in terms of database recovery. We summarize the main takeaway from the results in advance — **SAM generates databases that are closer to the original databases compared to PGM**. We conclude three major findings as follows.

(F4) SAM can well generalize to unseen queries while PGM cannot. From Table 5 and Table 6, we observe that SAM outperforms PGM across the board in terms of Q-Error on test query cardinality. On single-relation databases, SAM achieves significantly smaller error, with 500× less Mean error on Census and 100,000× less Mean error on DMV; on IMDB, SAM achieves at least two orders of magnitude smaller error at all percentiles. This is due to SAM's ability to process large-scale query workloads, and to capture much more information on the joint data distribution of the original database. Under a reasonable time limit, PGM can only process a very small number of training queries as discussed. As a consequence, this would greatly limit its ability to derive the underlying data distribution from the query workload.

(F5) The data distribution of the databases generated by SAM is closer to that of the original databases, compared to baseline methods. As shown in Table 7, across all three datasets, the relations generated by SAM achieve smaller cross entropy with regard to the original database relations. As SAM processes a much larger number of query constraints, it learns much more information on the joint data distribution, which SAM uses to generate a database that is closer to the original database.

(F6) Query latency of the databases generated by SAM is much closer to that of the original databases compared to baseline methods. From Table 8, we observe that for single-relation databases, SAM generates databases with smaller performance deviation of test queries compared to PGM, achieving at least 4×

Model	Census				DMV			
	Median	75th	90th	Mean	Median	75th	90th	Mean
PGM	46.00	872.0	3461	1097	646.0	$1 \cdot 10^5$	$1 \cdot 10^6$	$4 \cdot 10^5$
SAM	1.31	1.76	2.70	1.97	1.16	1.54	3.11	4.05

Table 5: Q-Error of test queries

Model	Median	75th	90th	Mean	Max
PGM	232.7	$6 \cdot 10^4$	$1 \cdot 10^6$	$9 \cdot 10^5$	$3 \cdot 10^7$
SAM w/o Group-and-Merge	38.67	$1 \cdot 10^5$	$3 \cdot 10^6$	$5 \cdot 10^6$	$3 \cdot 10^8$
SAM	2.29	5.39	27.78	2776	$2 \cdot 10^5$

Table 6: Q-Error of JOB-light queries on IMDB

Model	Census	DMV	IMDB
PGM	29.37	39.49	12.45
SAM	28.68	23.22	6.14

Table 7: Cross entropy of the generated relation

smaller performance deviation at all percentiles on Census. Table 9 shows that SAM's performance gain over baseline is even more significant on IMDB dataset. When measuring the performance deviation of a join query workload, *i.e.*, JOB-light, SAM achieves 20×, 80× and 40× smaller performance deviation at median, 75th and 90th percentile respectively. Due to the large performance deviation, previous database generators like PGM can hardly serve the use case of benchmarking and stress testing. In contrast, SAM is an ideal choice for those use cases.

Model	Census				DMV			
	Median	75th	90th	Mean	Median	75th	90th	Mean
PGM	1.38	2.67	3.86	1.81	145.2	610.0	798.4	311.4
SAM	0.26	0.54	1.05	0.43	103.0	339.7	630.0	221.8

Table 8: Performance deviation of test queries (millisecond)

Model	Median	75th	90th	Mean	Max
PGM	19.20	373.9	2637	1565	$3 \cdot 10^4$
SAM	0.89	4.86	65.75	121.0	5730

Table 9: Performance deviation of JOB-light queries (second)

5.5 Does the Group-and-Merge Algorithm help SAM for Join Key Assignment?

From Table 3, Table 4 and Table 6, we see that without the *Group-and-Merge* algorithm, the performance of SAM degrades in terms of both data fidelity and database recovery. The degradation in Q-Error is especially noticeable at the larger percentiles. This is because SAM w/o Group-and-Merge performs poorly on join queries involving three or more tables, as the join key assignment process only maintains the data correlation between pairs of primary key relation and foreign key relation. As a result, the generated base relations no longer recover the full outer join distribution learned by the model. Being able to take into account the joint distribution, *Group-and-Merge* helps assign join keys which recover the full outer join of the original database.

5.6 How Fast is SAM in Database Generation?

We then investigate the generation time of SAM, which refers to to the time it takes to sample from the model and perform the necessary post-processing steps to generate a synthetic database, *e.g.*, join key assignment.

Single-relation databases. The generation time of SAM on single-relation databases is very fast due to the batched computing on GPUs. In our experiments, on a GPU, SAM can generate the full database in seconds (1.2s) for Census and minutes (2.7mins) for DMV. The generation process can be even faster by using multiple GPUs. On the contrary, PGM requires 19 seconds and 0.9 hour to generate the Census and DMV database respectively using our single-threaded implementation on CPU, which is an order of magnitude slower. Although the generation process of PGM can also be accelerated through multi-threading or possibly a GPU implementation, significantly more work is required and it is unlikely to be faster than that of SAM.

Multi-relation databases. The generation time of SAM on multi-relation databases depends on the number of full outer join tuples it samples from the AR model. Figure 6 shows the plot of the generation time against the number of full outer join tuples sampled, as well as the median Q-Error of input query cardinalities. We observe that generation time scales linearly with regard to the number of full outer join tuples, while the median Q-Error plateaus after sampling around 120 million tuples. Thus, a high fidelity database can be generated with only 120 million samples for IMDB, which are only roughly 1/20,000 of the full outer join size. The time it takes for SAM to sample and process 120 millions tuples is around 1.2 hour, which is faster than the time it takes for PGM to generate the database (2.8 hour).

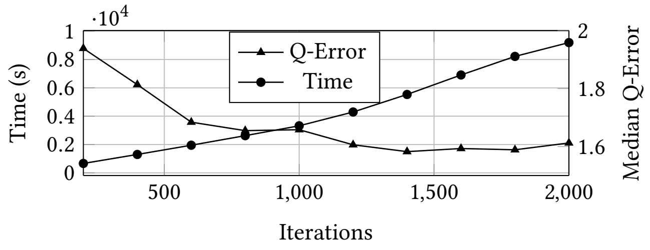

Figure 6: Generation time of SAM and the corresponding Q-Error on IMDB with varying sample sizes.

5.7 How Does the Performance of Database Recovery Scale with Query Workload Size?

SAM can scale to large-scale query workloads because its complexity is linear in the number of input cardinality constraints. Therefore, one interesting problem is whether the performance of SAM in database recovery improve with a larger query workload size. To answer this, we evaluate the cross entropy and Q-Error of test query cardinality with varying workload size (or the number of input cardinality constraints) from 20k to 100K. Note that all SAM trials finish training in the fixed time frame. As the results on other databases have the similar trend, we show the results on Census in Figure 7 due to the page limit. From the results, we can conclude two findings. First, a larger number of input cardinality constraints can provide more information on the underlying data distribution. Second, SAM can efficiently and effectively learn from the larger query workloads to recover the original database.

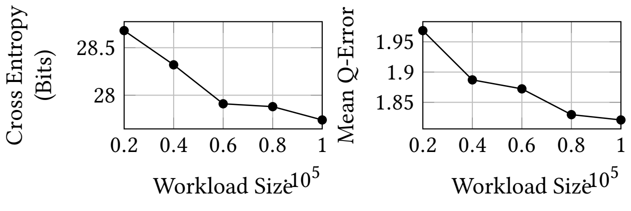

Figure 7: Performance of database recovery with varying workload size on Census.

5.8 How Does Workload Coverage impact the Performance of Database Recovery?

Ideally, SAM can recover the distribution of the original database if the query workload covers the entire data space. SAM's performance of database recovery may degrade when the query workload only covers a subspace of data. We study the impact of workload coverage on the performance of database recovery. We first synthesize equal-sized training workloads with different coverage ratio, *i.e.* the ratio between the size of the range covered by the query workload and the domain size of each column. We then generate a synthetic database from each workload and evaluate the performance of database recovery. The experiments are conducted on Census dataset.

From the results in Figure 8, we observe that as the coverage ratio increases, both the cross entropy of the generated databases and the mean Q-Error of the test query workload decreases. This shows that incomplete query coverage does have a negative impact on SAM's performance of database recovery. The lower the coverage ratio, the more degrading in the performance of database recovery.

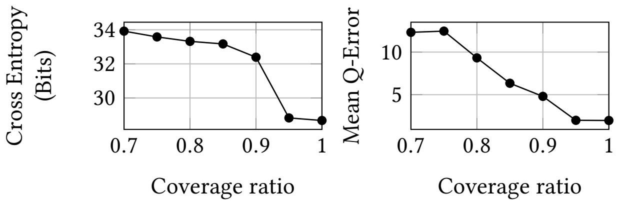

Figure 8: Performance of database recovery with varying workload coverage ratio on Census.

6 RELATED WORK

Database Generation from Queries. A number of works have studied the problem of database generation from queries [4, 5, 21–23]. QAGen [5] and MyBenchmark[23] proposed methods to generate database(s) satisfying the cardinality constraints of a parameterized query plan template for database testing. Touchstone [21] improved on the scalability of previous methods and proposed a parallel framework to generate large-scale databases. However, as these methods generate both the database(s) and the corresponding parameters, data characteristics cannot be specified through the queries, which limits the application of the methods to the problems of database benchmarking and stress testing. The Database Generation Problem was first defined in [4], and the PGM-based approaches were proposed for the problem, which we discussed in Section 2.3. One key difference between our work and [4] is that we are able to handle large-scale query workloads while PGM can only handle query workloads with very small number of queries as shown in our experiments, and our method can recover the data distribution of the original database by processing a large-scale query workload.

Database Generation from Data. The problem of database generation from data, or relational data synthesis, traces back to *privacy-preserving data publishing* [3, 6, 10, 20], which solves the problem of sharing data utility in a way that preserves sensitive and private information. With the advances in deep learning, a recent trend is to apply *generative adversarial networks (GAN)* to relational data synthesis [8, 11, 28, 30]. Our work differs from work on relational data synthesis as we assume no access to the database instance. Moreover, relational data synthesis usually focus on generating a single relation, while our work extends to the case of generating a full database comprising multiple relations.

Cardinality Estimation. This work is related to cardinality estimation [9]. First, we share the same input (a set of queries with cardinalities) with query-driven cardinality estimation [16, 26, 33]. However, its goal is to build a model to predict the cardinalities of incoming queries by learning from the observed queries, and thus most of the query-driven cardinality estimators are *discriminative* since they do not directly model the underlying data distribution. Consequently, we cannot sample data from these models, and thus they are not suitable for our work. The construction of SAM is based on UAE-Q [34]. Note that UAE-Q does not support database generation. Second, data-driven cardinality estimators [14, 35, 36] are constructed from the underlying data. This is different from our input settings.

7 CONCLUSION

This work considers the practical problem of generating a database from query workloads, such that the generated database resembles the original database in terms of data distribution. We propose SAM , which uses an AR model to learn the joint distribution of the full outer join from cardinality constraints through differentiable progressive sampling. SAM generates unbiased samples for base relations from the AR model through *inverse probability weighting* and *scaling*. We also devise the *Group-and-Merge* algorithm for join key assignment, so that the generated base relations can better recover the full outer join. Extensive experiments on real-world datasets demonstrate the superior performance of SAM in terms of generated data fidelity to input query constraints, closeness to the original database, and processing as well as generation efficiency compared to previous methods.

Avenues for future research include more effective algorithms for join key assignment. It is also interesting to study possible improvements to the differentiable progressive sampling algorithm.

ACKNOWLEDGEMENT

This research is supported in part by MOE Tier-2 grant MOE2019-T2-2-181 and MOE Tier-1 grant RG114/19(S). This research was conducted when Peizhi Wu was with NTU.

REFERENCES

[1] [n.d.]. Replaying a Database Workload, Oracle. https://docs.oracle.com/cd/E11882_01/server.112/e41481/dbr_replay.htm#RATUG131.

[2] [n.d.]. UCI machine learning repository. https://archive.ics.uci.edu/ml/index.php.

[3] Dakshi Agrawal and Charu C Aggarwal. 2001. On the design and quantification of privacy preserving data mining algorithms. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 247–255.

[4] Arvind Arasu, Raghav Kaushik, and Jian Li. 2011. Data generation using declarative constraints. In *SIGMOD*. 685–696.

[5] Carsten Binnig, Donald Kossmann, Eric Lo, and M Tamer Özsu. 2007. QAGen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 341–352.

[6] Justin Brickell and Vitaly Shmatikov. 2008. The cost of privacy: destruction of data-mining utility in anonymized data publishing. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 70–78.

[7] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2001. STHoles: a multidimensional workload-aware histogram. In *SIGMOD*. 211–222.

[8] Edward Choi, Siddharth Biswal, Bradley Malin, Jon Duke, Walter F Stewart, and Jimeng Sun. 2017. Generating multi-label discrete patient records using generative adversarial networks. In *Machine learning for healthcare conference*. PMLR.

[9] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. 2012. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Found. Trends Databases* 4, 1–3 (Jan. 2012), 1–294. https://doi.org/10.1561/1900000004

[10] Josep Domingo-Ferrer. 2008. A survey of inference control methods for privacy-preserving data mining. In *Privacy-preserving data mining*. Springer, 53–80.

[11] Ju Fan, Tongyu Liu, Guoliang Li, Junyou Chen, Yuwei Shen, and Xiaoyong Du. 2020. Relational data synthesis using generative adversarial networks: A design space exploration. *arXiv preprint arXiv:2008.12763* (2020).

[12] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. 2015. Made: Masked autoencoder for distribution estimation. In *International Conference on Machine Learning*. PMLR, 881–889.

[13] Lise Getoor, Benjamin Taskar, and Daphne Koller. 2001. Selectivity estimation using probabilistic models. In *SIGMOD*. 461–472.

[14] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *VLDB* 13, 7, 992–1005.

[15] Daniel G Horvitz and Donovan J Thompson. 1952. A generalization of sampling without replacement from a finite universe. *Journal of the American statistical Association* 47, 260 (1952), 663–685.

[16] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*.

[17] Daphne Koller and Nir Friedman. 2009. *Probabilistic graphical models: principles and techniques*. MIT press.

[18] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[19] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *The VLDB Journal* 27, 5 (2018), 643–668.

[20] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. 2007. t-closeness: Privacy beyond k-anonymity and l-diversity. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 106–115.

[21] Yuming Li, Rong Zhang, Xiaoyan Yang, Zhenjie Zhang, and Aoying Zhou. 2018. Touchstone: generating enormous query-aware test databases. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 575–586.

[22] Eric Lo, Nick Cheng, and Wing-Kai Hon. 2010. Generating databases for query workloads. *VLDB* 3, 1-2 (2010), 848–859.

[23] Eric Lo, Nick Cheng, Wilfred WK Lin, Wing-Kai Hon, and Byron Choi. 2014. MyBenchmark: generating databases for query workloads. *The VLDB journal* 23, 6 (2014), 895–913.

[24] Mohammad Ali Mansournia and Douglas G Altman. 2016. Inverse probability weighting. *Bmj* 352 (2016).

[25] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *PVLDB* 2, 1 (2009), 982–993.

[26] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2019. An Empirical Analysis of Deep Learning for Cardinality Estimation. *arXiv preprint arXiv:1905.06425* (2019).

[27] George Papamakarios, Theo Pavlakou, and Iain Murray. 2017. Masked autoregressive flow for density estimation. In *NIPS*. 2338–2347.

[28] Noseong Park, Mahmoud Mohammadi, Kshitij Gorde, Sushil Jajodia, Hongkyu Park, and Youngmin Kim. 2018. Data synthesis based on generative adversarial networks. *arXiv preprint arXiv:1806.03384* (2018).

[29] James M Robins, Andrea Rotnitzky, and Lue Ping Zhao. 1994. Estimation of regression coefficients when some regressors are not always observed. *Journal of the American statistical Association* 89, 427 (1994), 846–866.

[30] Saravanan Thirumuruganathan, Shohedul Hasan, Nick Koudas, and Gautam Das. 2019. Approximate query processing using deep generative models. *arXiv preprint arXiv:1903.10000* (2019).

[31] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. 2011. Lightweight graphical models for selectivity estimation without independence assumptions. *PVLDB* 4, 11 (2011), 852–863.

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[33] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a learning optimizer for shared clouds. *VLDB* 12, 3 (2018), 210–222.

[34] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation. In *SIGMOD*. 2009–2022.

[35] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2021. NeuroCard: One Cardinality Estimator for All Tables. *PVLDB* (2021).

[36] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2020. Deep Unsupervised Cardinality Estimation. *VLDB* 13, 3, 279–292.

[37] Federico Zanettin. 2019. State of New York. Vehicle, snowmobile, and boat registrations. catalog.data.gov/dataset/vehicle-snowmobile-and-boat-registrations.