Data Generation using Declarative Constraints

Arvind Arasu
Microsoft Research
Redmond, WA
arvinda@microsoft.com

Raghav Kaushik
Microsoft Research
Redmond, WA
skaushi@microsoft.com

Jian Li
University of Maryland
College Park, MD
lijian@cs.umd.edu

ABSTRACT

We study the problem of generating synthetic databases having declaratively specified characteristics. This problem is motivated by database system and application testing, data masking, and benchmarking. While the data generation problem has been studied before, prior approaches are either non-declarative or have fundamental limitations relating to data characteristics that they can capture and efficiently support. We argue that a natural, expressive, and declarative mechanism for specifying data characteristics is through *cardinality constraints*; a cardinality constraint specifies that the output of a query over the generated database have a certain cardinality. While the data generation problem is intractable in general, we present efficient algorithms that can handle a large and useful class of constraints. We include a thorough empirical evaluation illustrating that our algorithms handle complex constraints, scale well as the number of constraints increase, and outperform applicable prior techniques.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms, Performance, Reliability, Experimentation

Keywords

Data Generation, Testing, Masking, Benchmarking, Constraints

1. INTRODUCTION

We consider the problem of generating a *synthetic* database instance having certain data characteristics. Many applications require synthetically generated data:

1. *DBMS testing*: When we design a new DBMS component such as a new join operator or a new memory manager, we require synthetic database instances with specific characteristics to test correctness and performance of the new component [7, 22]. For example, to test the code module of a hybrid hash join that handles spills to disk, we might need a database instance with a high *skew* on the outer join attribute. As another example, to study the interaction of the memory manager and multiple hash join operators, we might need a database instance that has particular intermediate result cardinalities for a given query plan [9].

2. *Data masking and database application testing*: Organizations sometimes outsource the testing of their database applications to other organizations. However an outsourcing organization might not be able to share its internal databases (over which the applications run) with the testing organization due to privacy considerations, requiring us to generate a synthetic database that behaves like the original database for the purposes of testing. (We emphasize that our goal here is not to study the general *data masking* problem with its privacy considerations; we are merely suggesting that data generation might be a useful component of a general data masking solution.)

3. *Benchmarking*: In order to decide between multiple competing data management solutions, a customer might be interested in *benchmarking* the solutions [22]. The standard benchmarks such as TPC-H might not capture many of the application scenarios and data characteristics of interest to the customer, motivating the need for synthetic data generation. A related scenario is *upscaling*, where we are interested in generating a synthetic database that is an upscaled version of an existing database. Upscaling is useful for future capacity planning purposes.

Data characteristics and cardinality constraints: The applications of data generation above require a wide variety of data characteristics in the generated synthetic databases. A natural class of characteristics are schema properties such as key and referential integrity constraints, functional dependencies, and domain constraints (e.g., *age* is an integer between 0 and 120). A synthetic database for DB application testing often needs to satisfy such constraints since the application being tested might require these constraints for correct functioning. If DB application testing involves a visual component with a tester entering values in fields of a form, the synthetic database might need to satisfy *natu-*

SIGMOD'11, June 12–16, 2011, Athens, Greece.

ralness properties, e.g., the values in an address field should "look like" real addresses.

In benchmarking and DBMS testing, we typically need to capture characteristics that can influence the performance of a query over the generated database. These include, for example, ensuring that values in a column be distributed in a particular way, ensuring that values in a column have a certain skew, or ensuring that two or more columns are *correlated*. We note correlations can involve joining multiple tables. For example, in a customer-product-order database, we might need to capture correlations between the age of customers and the category of products they purchase. In data masking, we might require synthetic data to result in the same application performance as the original data, without revealing sensitive information from the original data.

In addition to the richness of data characteristics, applications might require *several* properties and constraints be together satisfied in a generated database. This requirement motivates the need for a *declarative* approach to data generation as opposed procedural approaches considered in [7, 15]. As a concrete example, consider generating a customer-product-order database where we need to capture correlations between several pairs of columns such as customer age and product category, customer age and income, and product category and supplier location. It is fairly nontrivial for a programmer to design a procedure that outputs a database with all of the above properties, even with the right procedural primitives.

A natural and expressive language for specifying data characteristics is a *set* of *cardinality constraints*. A cardinality constraint specifies that the output of a given query over the generated database should have a particular cardinality. As a simple example, we can (approximately) specify the distribution of values in a column by providing a histogram, and a histogram can be represented as a collection of cardinality constraints, one for each bucket. In Section 2, we show that many of the data characteristics discussed earlier can be represented using cardinality constraints.

The idea of using cardinality constraints for data generation is not new and has been proposed in QAGen [6] and its extension MyBenchmark [22]. However, in this work cardinality constraints are mostly used for capturing workload characteristics and the ability of cardinality constraints to express more general data characteristics is not discussed.

Motivated by the above discussion, the goal of this paper is to design efficient algorithms for generating synthetic databases that satisfy a given set of cardinality constraints. The set of constraints provided as input can be large (say, thousands); for example, even specifying a simple histogram can require 10s or 100s of constraints. The queries in the constraints can be complex, possibly involving joins over multiple tables.

Prior Work: While QAGen [6] and MyBenchmark [22] do not discuss the expressiveness aspects of cardinality constraints, their techniques are quite general and can be used for our purposes. However, they have some basic limitations. QAGen and MyBenchmark assume that cardinality constraints are available in a particular form called *annotated query plans (AQP)*. An annotated query plan is a query plan with a subset of plan nodes annotated with cardinalities. We can show that we can encode cardinality constraints as AQPs and vice-versa. For data generation, QAGen uses a novel approach called *symbolic query processing*. Briefly, it starts with a symbolic database; a symbolic database is like a regular database, but its attribute values are symbols (variables), not "constants." It then translates the input AQPs to constraints over the symbols in the database, and invokes a black-box *constraint satisfaction program (CSP)* to identify values for symbols that satisfy all the constraints.

One limitation of QAGen is that it can handle a single AQP, and therefore cannot be directly used to generate databases that satisfy multiple arbitrary constraints. This limitation is identified and addressed in MyBenchmark [22]. Briefly, to handle n AQPs, MyBenchmark uses QAGen to generate n symbolic databases with constraints and performs "matching" between these databases to heuristically identify $m \leq n$ databases that together satisfy all the AQPs. MyBenchmark is not guaranteed to produce a single database instance and this functionality can be unsuitable for some applications requiring synthetic data. For example, we cannot use multiple database instances for DB application testing, since no single instance reflects all the characteristics of the original database.

One advantage of using a general purpose CSP is that it enables QAGen to handle complex queries, e.g., queries with HAVING clauses. However, this generality comes with a performance cost. The number of times QAGen and MyBenchmark invoke a CSP grows with the size of the generated database and this has serious performance implications as the experiments in [6, 22] indicate.[1] The algorithms that we propose do not have these limitations: they always generate a single database instance and their dependence on the generated database size is limited to the cost of materializing the database.

Interestingly, recent work on cardinality estimation using *maximum entropy principle* [27, 28] can be adapted to derive algorithms for data generation, and we discuss this possibility in detail in Section 4. However, briefly, cardinality estimation using maximum entropy is known to be a very hard problem and adaptations of current solutions do not efficiently handle complex constraints.

Summary of Contributions: We formally introduce cardinality constraints in Section 2 and show that a set of cardinality constraints forms an expressive language for specifying data characteristics. In Section 3, we state the formal data generation problem and show that the general problem is *NEXP-complete* and therefore hard. We present our algorithms in Section 4. While the general data generation problem is hard, our algorithms are able to handle a large and useful class of constraints. Our algorithms are *probabilistically approximate*, meaning that they satisfy all constraints in expectation. We note that this is sufficient for most applications of data generation. Our algorithms are also sensitive to the complexity of the input cardinality constraints in a precisely quantifiable way and use ideas from *probabilistic graphical models* [26]. We include detailed experimental evaluation of our algorithms in Section 6 and conclude.

[1] One aspect of QAGen that we do not consider in this paper is *parameters* in AQPs. In the full version of the paper, we show that a data generation problem instance having cardinality constraints with parameters can be transformed to an instance not involving parameters, for a large class of constraints.

2. CARDINALITY CONSTRAINTS

In this section, we formally introduce cardinality constraints and discuss their expressiveness. We need a few notations first. We denote a relation R with attributes $A_1, \ldots, A_n$ as $R(A_1, \ldots, A_n)$. We use $Attr(R) = \{A_1, \ldots, A_n\}$ to denote the set of attributes of relation R. A database $\mathcal{D}$ is a collection of relations $R_1, \ldots, R_l$. Given the schema of $\mathcal{D}$, a *cardinality constraint* is of the form

$$|\pi_{\mathcal{A}}\sigma_P(R_{i_1} \bowtie \cdots \bowtie R_{i_p})| = k$$

where $\mathcal{A}$ is a set of attributes, P is a selection predicate, and k is a non-negative integer. A database instance *satisfies* a cardinality constraint if evaluating the relational expression over the instance produces k tuples in the output. Throughout this paper, we assume that relations are bags[2] and relational operators use bag semantics. The projection operator in cardinality constraints is *duplicate eliminating*; the input and output cardinalities of a *duplicate preserving projection* operator are identical, and therefore duplicate preserving projections are not interesting in constraints. (The projection operator is optional.)

We now show that a set of a cardinality constraints can be used to declaratively encode various data characteristics of interest.

Schema Properties:[3] We can specify that a set of attributes $\mathcal{A}_k \subseteq Attr(R)$ is a key of R using two constraints $|\pi_{\mathcal{A}_k}(R)| = N$ and $|R| = N$. We can specify that $R.A$ is a foreign key referencing $S.B$ using the constraints $|\pi_A(R \bowtie_{A=B} S)| = N$ and $|R| = N$. We can similarly represent more general inclusion dependencies between attribute values of one table and attribute values of another. Such inclusion dependencies can also be used with reference "knowledge" tables such as a table of all US addresses to ensure that the generated databases satisfy various naturalness properties.

Value distributions: As mentioned earlier, we can approximately capture the value distribution of a column using a histogram. We can specify a single dimension histogram by including one constraint for each histogram bucket. The constraint corresponding to the bucket with boundaries $[l, h]$ having k tuples is $|\sigma_{l \leq A \leq h}(R)| = k$. We can capture correlations between attributes using multi-dimension histograms such as STHoles [8], which can again be encoded using one constraint for each histogram bucket. Correlations spanning multiple tables can be specified using joins and multi-dimension histograms. For example, we can specify correlations between customer age and product category in a customer-product-orders database using multi-dimension histograms over the view ($Customer \bowtie Orders \bowtie Product$). We can approximately constrain the performance of a query plan over generated data by specifying intermediate cardinalities as shown in Figure 1. Each intermediate cardinality maps to a cardinality constraint. In the data masking scenario from Section 1, these intermediate cardinalities can be obtained by evaluating the query plan on the original data to ensure that the performance of the plan on original and synthetic data are similar. (We have not fully explored the privacy related issues in this setting. We note that cardinality constraints integrate nicely with differential privacy [13];

[2]A bag is a multi-set where an element can appear multiple times.

[3]Although, we can encode schema properties using general cardinality constraints, for efficiency purposes, our algorithms handle them in a special way, different from other constraints.

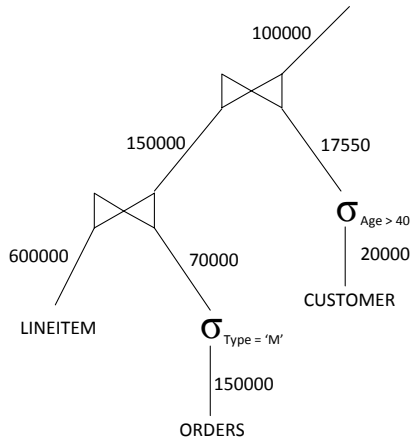

Figure 1: Query Plan intermediate cardinalities

instead of using actual intermediate cardinalities, we can fudge cardinalities using differential privacy algorithms and use the fudged constraints for data generation. We are currently exploring these directions.) The full version of the paper includes examples of using cardinality constraints to capture more complex attribute correlations such as those of [14], join distributions between relations, and skew of values in a column.

We view cardinality constraints as a useful programmatic abstraction for data generation and we envision automated techniques, not manual approaches, generating the constraints. The techniques for generating constraints are application specific and outside the scope of this work.

We note that the set of cardinality constraints is complete, in a sense that any database instance can be fully specified using such constraints (for each tuple, we use a constraint to specify its existence). Also, the cardinality constraints that we consider in this paper are a special class of *logic programs and cardinality constraints* studied in [30]; the focus of this work is to extend logic programs with cardinality constraints and establish formal semantics.

3. PROBLEM FORMULATION

We now formally state the data generation problem:

Data Generation Problem (DGP): Given a database schema and a collection of cardinality constraints $C_1, \ldots, C_m$, generate a database instance conforming to the schema that satisfies all the constraints.

In the decision version of the problem, the output is YES if there exists a database instance that satisfies all the constraints and NO, otherwise. We can show even the decision version of the problem is extremely hard.

THEOREM 1. *The decision version of the data generation problem is NEXP-complete.*

Due to space constraints, we defer all proofs to the full version of the paper. We note that the problem of checking whether a logical program with cardinality constraints has a model is NEXP-complete [30], but Theorem 1 does not follow from the results in [30] since logical programs of [30] are more general than the cardinality constraints we consider.

While the general DGP problem is hard, there exist probabilistically approximate efficient algorithms for a large and

useful class of constraints that we present next. These algorithms satisfy the input constraints only in *expectation*; as we observed in Section 1, such approximation is usually acceptable in practice.

4. ALGORITHMS

We begin (in section 4.1) by presenting an algorithm for the simple case of a single table with a single attribute that involves solving a linear program (LP). A straightforward generalization of the LP approach for multiple attributes produces an exponentially large LP, and we use ideas from from probabilistic graphical models to reduce the size of the LP. In section 4.3, we present algorithms for generating multiple tables possibly involving join constraints. Sections 4.1-4.3 assume that the input constraints do not involve projections. We discuss constraints with projections in section 4.4.

We first present some notation and simplifying assumptions. We denote the domain of attribute A using $Dom(A)$. We assume the domains of all attributes are positive integers; this assumption is without loss of generality since values from other domains can be 1-1 mapped to positive integers. To simplify presentation, we further assume that the domain of all attributes is $[D] = \{1, \ldots, D\}$ for a known positive integer D. Removing the assumption that D be known in advance and handling different domains for different attributes is straightforward.

For presentation simplicity, we assume selection predicates are conjunctions of range predicates of the form $A \in [l, h]$; equality predicate is a special case where $l = h$. Our algorithms can be extended to work with selection predicates with disjunction and non-equalities such as $\geq, \leq, >$, and $<$.

In the rest of this section, we use $C_1, \ldots, C_m$ to denote the input constraints. We specify the exact form of the constraints in each subsection.

4.1 Single Table, Single Attribute

Let $R(A)$ denote the table being generated. Without loss of generality, each constraint C_j $(1 \leq j \leq m)$ is in the canonical form $|\sigma_{l_j \leq A < h_j}(R)| = k_j$. We generate a simple *integer linear program (ILP)* to solve this instance of DGP. For each $i \in [D]$ we create a variable x_i that represents the number of copies of i in R. We create the following system of equations to capture the m constraints:

$$\sum_{i=l_j}^{h_j-1} x_i = k_j \quad \text{for} \quad j = 1, \ldots, m$$

We further require that each x_i is a nonnegative integer. We can show that any solution to the above ILP corresponds to a solution of the DGP instance.

In general, solving an ILP is NP-hard [19]. However, the above ILP has a special structure: we can show that the matrix corresponding to the system of equations has a property called *unimodularity* [19]. This property implies that a solution of the corresponding *linear programming (LP) relaxation* is integral[4]. (We obtain the LP relaxation by dropping the integer requirement.) An LP can be solved in polynomial time, but this does not imply a polynomial time solution to DGP since the number of variables in the LP is proportional to domain size D, which may be much larger than the sizes of the input and the database instance being output. We next present a simple *intervalization* trick to reduce the size of the LP.

Intervalization: Let $v_1 = 1, v_2, \ldots, v_l = D + 1$ denote in increasing order the distinct constants occurring in predicates of constraints C_j including constants 1 and $D+1$. We define $(l-1)$ *basic intervals* $[v_i, v_{i+1})$ $(1 \leq i < l)$. We introduce a variable $x_{[v_i, v_{i+1})}$ for each basic interval $[v_i, v_{i+1})$ to represent the number of tuples in $R(A)$ that belong to the interval. Consider a constraint $C_j : |\sigma_{l_j \leq A < h_j}(R)| = k_j$. By construction, there exist $v_p = l_j$ and $v_q = r_j$, and we use the following equation to capture C_j:

$$\sum_{i=p}^{q-1} x_{[v_i, v_{i+1})} = k_j$$

As before, a solution to the above LP can be used to construct a solution for the DGP instance. We can easily see that the number of variables is at most twice of the number of constraints, implying a polynomial time solution[5] to the DGP problem.

EXAMPLE 1. *Consider a DGP instance with three constraints* $|\sigma_{20 \leq A < 60}(R)| = 30$, $|\sigma_{40 \leq A < 101}(R)| = 40$, *and* $|R| = 50$ *and assume a domain size* $D = 100$. *There are* 4 *basic intervals:* $[1, 20)$, $[20, 40)$, $[40, 60)$, $[60, 101)$. *The corresponding linear program consists of the three equations:*

$$x_{[1,20)} + x_{[20,40)} + x_{[40,60)} + x_{[60,101)} = 50$$
$$x_{[20,40)} + x_{[40,60)} = 30$$
$$x_{[40,60)} + x_{[60,101)} = 40$$

One solution to the LP is $x_{[1,20)} = 2$, $x_{[20,40)} = 8$, $x_{[40,60)} = 22$, *and* $x_{[60,101)} = 18$. *To generate* $R(A)$, *we pick* 2 *values (e.g., at random) from* $[1, 20)$, 8 *values from* $[20, 40)$, *and so on.*

This intervalization trick is applicable and implicitly part of all of our algorithms, although for presentation simplicity we may not explicitly mention this fact.

4.2 Single Table, Multiple Attributes

Let $R(A_1, \ldots, A_n)$ denote the table being generated. Each constraint C_j is of the form $|\sigma_{P_j}(R)| = k_j$. For conciseness, we sometimes denote constraint C_j as the pair $\langle P_j, k_j \rangle$ in this subsection. The following theorem implies that the DGP problem becomes significantly harder when we move from single to multiple attributes.

THEOREM 2. *The decision version of the single table data generation problem without projections is NP-complete for even two attributes* $(n = 2)$.

We first consider a generalization of the algorithm in Section 4.1. For every tuple $t \in Dom(A_1) \times \cdots \times Dom(A_n)$, we create a variable x_t representing the number of copies of t in R. For each constraint $C_j = \langle P_j, k_j \rangle$, we generate a linear equation:

$$\sum_{t : P_j(t) = \text{true}} x_t = k_j$$

[4]This solution is integral only in the presence of a linear optimization criterion, so we need to add a dummy criterion to get integral solutions.

[5]Given an LP solution, the time for generating the actual table is linear in the size of the output, which may be independent of the input size. However, since any algorithm takes that much time, we do not count it in the running time.

With LP relaxation, a solution to the above set of equations might not always be integral; otherwise, Theorem 2 would imply $P = NP$. However, slightly violating some cardinality constraints is acceptable for most applications of data generation. We can derive a probabilistically approximate solution by starting with an LP relaxation solution and performing randomized rounding: Round x_t to $\lfloor x_t \rfloor$ with probability $x_t - \lfloor x_t \rfloor$ and to $\lceil x_t \rceil$ with probability $\lceil x_t \rceil - x_t$. We can prove that relation R generated in this manner satisfies all constraints in expectation: $\mathbb{E}[|\sigma_{P_j}(R)|] = k_j$ for all constraints C_j. We refer to this algorithm as LPAlg.

Even with intervalization, the number of variables created by LPAlg can be exponential in the number of attributes. We next present more sophisticated algorithms that use ideas from graphical models [26]. If the input constraints are low-dimensional (involve a small number of attributes) and sparse—this is often the case in practice—these algorithms significantly outperform LPAlg.

4.2.1 Algorithms based on Graphical Models

We begin with a toy example illustrating a more efficient strategy for data generation than LPAlg.

Example 2. *Consider a DGP instance with domain size* $|D| = 2$ *and* $2n + 1$ *constraints* $|R| = 1000$, $|\sigma_{A_i=1}(R)| = 500$ *and* $|\sigma_{A_i=2}(R)| = 500$ $(1 \leq i \leq n)$. LPAlg *solves an LP involving* 2^n *variables for this instance. There exists a simpler strategy: We generate* 1000 *random tuples, where each tuple is generated by picking each of its attribute values from* $\{1, 2\}$ *uniformly at random. It is easy to see that this generated instance satisfies all constraints in expectation.*

Informally, the strategy in Example 2 "decouples" attributes from one another and generates values for each attribute "independently." The algorithms we present next are essentially generalizations of this idea that identify and exploit various "independence" properties between attributes for efficient data generation.

Figure 2 presents a general class of algorithms, which includes the algorithm of Example 2. We assume for simplicity that the size of R, $|R| = N$, is provided as input. With each attribute A_i, we associate a random variable X_i that assumes values in $Dom(A_i)$. Let $\mathcal{X} = \{X_1, \ldots, X_n\}$. We denote a joint probability distribution over $X_1, \ldots, X_n$ using $p(X_1, \ldots, X_n)$ (or $p(\mathcal{X})$, for short). In the first step, we identify a distribution $p(\mathcal{X})$ that belongs to a special class called *generative distributions*.

Definition 1. *(Generative Distribution) A generative distribution is a probability distribution* $p(\mathcal{X})$ *that satisfies the property that for each constraint* $C_j = \langle P_j, k_j \rangle$*, the probability that predicate* P_j *is true for a tuple sampled from* $p(\mathcal{X})$ *is* k_j/N.

In the next step, we independently sample N times from this distribution to generate an instance R. By construction, this instance satisfies all constraints in expectation, i.e., $\mathbb{E}[|\sigma_{P_j}(R)|] = k_j$. Moreover, using Hoeffding's inequality, we can show that the probability mass is concentrated around the mean:

$$\Pr[||\sigma_{P_j}(R)| - k_j| \geq t] \leq 2\exp(-\frac{2t^2}{N}).$$

Example 3. *The algorithm in Example 2 uses the uniform generative distribution* $p(x_1, \ldots, x_n) = 1/2^n$ *for all* $x_1, \ldots, x_n$.

Input: A data generation problem involving $R(A_1, \ldots, A_n)$ and constraints $C_1, \ldots, C_m$ and $|R| = N$
1. Identify a generative probability distribution $p(\mathcal{X})$
2. Sample N times from $p(\mathcal{X})$ to generate R

Figure 2: A general probabilistic algorithm

The following proposition asserts that the algorithm of Figure 2 is always feasible:

Proposition 3. *If an instance of single table data generation problem (without projections) has a solution, there exists a generative probability distribution for the instance.*

We next discuss the problem of identifying a generative probability distribution $p(\mathcal{X})$. In general, there could be several generative probability distributions for an instance of single table DGP. Theorem 4 states there always exists a generative distribution that *factorizes* into a product of simpler functions; we later use this factorization to identify such a distribution. We first introduce some notation used in the statement of Theorem 4. For each constraint $C_j = \langle P_j, k_j \rangle$, we use $Attrs(C_j)$ to denote the set of attributes appearing in predicate P_j and $\mathcal{X}(C_j)$ to denote the set of random variables corresponding to these attributes. For example, if C_j is $|\sigma_{A_1=5 \wedge A_3=4}(R)| = 10$, then $Attrs(C_j) = \{A_1, A_3\}$ and $\mathcal{X}(C_j) = \{X_1, X_3\}$. For any $\mathcal{X}' \subseteq \mathcal{X}$, we use $f(\mathcal{X}')$ to denote a function f over random variables in $\mathcal{X}'$. Function $f(\mathcal{X}')$ maps an assignment of values to random variables in $\mathcal{X}'$ to its range (which is usually nonnegative reals $\mathbb{R}^{\geq 0}$). We assume that each attribute A_i appears in at least one constraint; otherwise, we add the trivial constraint $|\sigma_{1 \leq A_i \leq D}(R)| = N$.

Theorem 4. *If an instance of single table data generation problem (without projections) has a solution, then there exists a generative probability distribution* $p(\mathcal{X})$ *that factorizes as:*

$$p(\mathcal{X}) = \prod_{\mathcal{X}_i : \exists C_j s.t. \mathcal{X}_i = \mathcal{X}(C_j)} f_i(\mathcal{X}_i)$$

for some functions f_i.

Example 4. *Consider a DGP instance where* $Attrs(C_1) = \{A_1, A_2\}$ *and for all other constraints* C_j $(j \neq 1)$, $|Attrs(C_j)| = 1$. *Theorem 4 asserts that there exists a generative probability distribution* $p(X_1, \ldots, X_n)$ *for this instance that can be expressed as* $f_1(X_1, X_2)f_3(X_3) \cdots f_n(X_n)$*, where* f_i *are some functions. Note that a DGP instance can have several generative distributions and all such distributions need not factorize as above, but there exists at least one that does.*

The factorization of a probability distribution implies various independence properties of the distribution. It is convenient to use an undirected graph to infer independence properties implied by a factorization [26]. The *Markov network* of a DGP instance is an undirected graph $G = (\mathcal{X}, E)$ with vertices corresponding to the random variables $X_1, \ldots, X_n$. Graph G contains an edge (X_i, X_j) whenever $\{X_i, X_j\} \subseteq \mathcal{X}(C_j)$ for some constraint C_j. The following lemma characterizes the independence properties of distributions $p(\mathcal{X})$ that factorize according to Theorem 4. It follows from a simple application of the well-known Hammersley-Clifford theorem [16, 26].

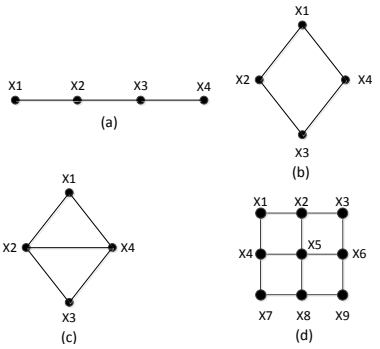

Figure 3: Example Markov Networks

LEMMA 1. *Let $\mathcal{X}_A, \mathcal{X}_B, \mathcal{X}_C \subseteq \mathcal{X}$ be nonoverlapping sets such that in the Markov network G every path from a vertex in $\mathcal{X}_A$ to a vertex in $\mathcal{X}_B$ goes through a vertex in $\mathcal{X}_C$. Then for any probability distribution that factorizes according to Theorem 4, $(\mathcal{X}_A \perp \mathcal{X}_B | \mathcal{X}_C)$.*

(The notation $(\mathcal{X}_A \perp \mathcal{X}_B | \mathcal{X}_C)$ denotes that $\mathcal{X}_A$ and $\mathcal{X}_B$ are conditionally independent given $\mathcal{X}_C$.) As a special case of Lemma 1, if $\mathcal{X}_A$ and $\mathcal{X}_B$ belong to different connected components then $\mathcal{X}_A \perp \mathcal{X}_B$ (unconditionally) for any distribution $p(\mathcal{X})$ that factorizes according to Theorem 4.

EXAMPLE 5. *Continuing Example 4, the Markov network for this instance has n vertices and a single edge (X_1, X_2). Lemma 1 implies that there exists a distribution $p(\mathcal{X})$ for which $(X_i \perp X_j)$ for all pairs $\{X_i, X_j\} \neq \{X_1, X_2\}$. Using chain rule and the above independences, we can show that $p(\mathcal{X}) = p(X_1, X_2)\ p(X_3) \cdots p(X_n)$, where, for example, $p(X_1, X_2)$ denotes the marginal distribution of (X_1, X_2). The algorithm we present next uses this observation to divide the problem of identifying $p(\mathcal{X})$ into "smaller" problems of identifying the marginals $p(X_1, X_2)$, $p(X_3), \ldots, p(X_n)$.*

EXAMPLE 6. *Consider a DGP instance whose Markov network is the path graph shown in Figure 3(a). There exists a generative $p(\mathcal{X})$ for which $(X_1 \perp X_3 | X_2)$, since the only path from X_1 to X_3 passes through X_2. For the graph of Figure 3(b), $(X_1 \perp X_3 | X_2, X_4)$, but $(X_1 \not\perp X_3 | X_2)$.*

We now present two algorithms for identifying a generative distribution. While the algorithms are easy to state, the rationale for some steps of the algorithms and their formal correctness depend on nontrivial properties of graphical models and are presented in the full version of the paper. The two algorithms are incomparable and depending on the input one can outperform the other.

Algorithm based on Chordal graphs

The first algorithm is designed based on the insight that the factors f_i in Theorem 4 allow a natural probabilistic interpretation if the Markov network is a *chordal graph*[6]. A graph is chordal if each cycle of length 4 or more has a *chord*; a chord is an edge joining two nonadjacent nodes of a cycle. The graph in Figure 3(b) is not chordal, but adding the edge (X_2, X_4) results in the chordal graph shown in Figure 3(c).

The formal algorithm is presented in Figure 4. We begin by constructing (Step 1) the Markov network of the input

[6]In the language of graphical models, the distribution $p(\mathcal{X})$ is a *decomposable* distribution if the Markov network is chordal [26].

INPUT: A data generation problem involving $R(A_1, \ldots, A_n)$ and constraints $C_1, \ldots, C_m$ and $|R| = N$
OUTPUT: A generative probability distribution $p(\mathcal{X})$
1. Construct the Markov network $G = (\mathcal{X}, E)$
2. Add edges to G to get a chordal graph $G_c = (\mathcal{X}, E_c)$
3. Identify maximal cliques $\mathcal{X}_{c1}, \ldots, \mathcal{X}_{cl}$ in G_c
4. Solve for marginal distributions $p(\mathcal{X}_{c1}), \ldots, p(\mathcal{X}_{cl})$
5. Use chordal graph property to construct $p(\mathcal{X})$ from the marginals $p(\mathcal{X}_{c1}), \ldots, p(\mathcal{X}_{cl})$

Figure 4: Indentifying a generative distribution using Chordal graphs

DGP instance. In general, the Markov network of a DGP instance need not be chordal. In Step 2, we use the algorithm of Tarjan and Yannakakis [31] to convert the Markov network $G = (\mathcal{X}, E)$ to a chordal graph $G_c = (\mathcal{X}, E_c)$, $E \subseteq E_c$, by adding additional edges if necessary. In Step 3, we identify the maximal cliques $\mathcal{X}_{c1}, \ldots, \mathcal{X}_{cl}$ of G_c. In Step 4, we "solve" for the marginal distributions $p(\mathcal{X}_{c1}), \ldots, p(\mathcal{X}_{cl})$; this step is discussed in detail later. Chordal graphs have a special property [26] that allows us to construct (in Step 5) the desired distribution $p(\mathcal{X})$ using the marginals $p(\mathcal{X}_{c1}), \ldots, p(\mathcal{X}_{cl})$. The following example illustrates this property.

EXAMPLE 7. *Consider a DGP instance whose Markov network is the path graph of Figure 3(a) and let $p(X_1, X_2, X_3, X_4)$ be a distribution that factorizes according to Theorem 4. The graph has no cycles and is therefore chordal with maximal cliques $\{X_1, X_2\}$, $\{X_2, X_3\}$, and $\{X_3, X_4\}$. We show that $p(X_1, X_2, X_3, X_4)$ can be computed using the marginals over these cliques:*

$$\begin{aligned} p(X_1, X_2, X_3, X_4) &= p(X_1, X_2) p(X_3 | X_1, X_2) p(X_4 | X_1, X_2, X_3) \\ &= p(X_1, X_2) p(X_3 | X_2) p(X_4 | X_3) \\ &= p(X_1, X_2) \frac{p(X_2, X_3)}{p(X_2)} \frac{p(X_3, X_4)}{p(X_3)} \end{aligned}$$

The second step follows from the first using the independence properties described in Example 6. $p(X_2)$ and $p(X_3)$ can be obtained from $p(X_2, X_3)$ by summing out X_3 and X_2, respectively. Finally, note that such a distribution is easy to sample from: we first pick x_1, x_2 from $p(X_1, X_2)$, then pick x_3 from $p(X_3 | X_2 = x_2)$ and then pick x_4 from $p(X_4 | X_3 = x_3)$.

To identify the marginal distributions $p(\mathcal{X}_{c1}), \ldots, p(\mathcal{X}_{cl})$ (Step 4, Figure 4), we construct and solve a system of linear equations. The variables in these equations are probability values $p(\mathbf{x})$, $\mathbf{x} \in Dom(\mathcal{X}_{ci})$ of the distributions $p(\mathcal{X}_{ci})$. In the following, we use $p_{\mathcal{X}_{ci}}$ to denote the marginal over variables $\mathcal{X}_{ci}$. The first set of equations below ensures that $p_{\mathcal{X}_{ci}}$ are valid probability distributions. The second set ensures that the marginal distributions satisfy all constraints within their "scope".

$$\sum_{\mathbf{x} \in Dom(\mathcal{X}_{ci})} p_{\mathcal{X}_{ci}}(\mathbf{x}) = 1 \qquad 1 \leq i \leq l$$

$$\sum_{\mathbf{x} \in Dom(\mathcal{X}_{ci}) : P_j(\mathbf{x}) = \text{true}} p_{\mathcal{X}_{ci}}(\mathbf{x}) = k_j / N \qquad \mathcal{X}(C_j) \subseteq \mathcal{X}_{ci}$$

Consider any two cliques $\mathcal{X}_{ci}$ and $\mathcal{X}_{cj}$ such that $\mathcal{X}_{ci} \cap \mathcal{X}_{cj} \neq \phi$. For any $\mathbf{x} \in Dom(\mathcal{X}_{ci} \cap \mathcal{X}_{cj})$, let $Ext_{\mathcal{X}_{ci}}(\mathbf{x})$ denote the set of assignments to $\mathcal{X}_{ci}$ that is consistent with the assignment $\mathbf{x}$.

INPUT: A data generation problem involving $R(A_1, \ldots, A_n)$ and constraints $C_1, \ldots, C_m$ and $|R| = N$
OUTPUT: A generative probability distribution $p(\mathcal{X})$
1. Construct the Markov network $G = (\mathcal{X}, E)$
2. Identify maximal cliques $\mathcal{X}_{c1}, \ldots, \mathcal{X}_{cl}$
3. Solve for marginal distributions $p(\overline{\mathsf{M}}(\mathcal{X}_{c1})), \ldots, p(\overline{\mathsf{M}}(\mathcal{X}_{cl}))$
4. Construct $p(\mathcal{X})$ from the marginals $p(\overline{\mathsf{M}}(\mathcal{X}_{c1})), \ldots, p(\overline{\mathsf{M}}(\mathcal{X}_{cl}))$

Figure 5: Indentifying a generative distribution using Markov Blankets

We include the following equation for each $\mathbf{x} \in Dom(\mathcal{X}_{ci} \cap \mathcal{X}_{cj})$ in the linear program.

$$\sum_{\mathbf{y} \in Ext_{\mathcal{X}_{ci}}(\mathbf{x})} p_{\mathcal{X}_{ci}}(\mathbf{y}) = \sum_{\mathbf{z} \in Ext_{\mathcal{X}_{cj}}(\mathbf{x})} p_{\mathcal{X}_{cj}}(\mathbf{z})$$

The marginal distribution $p(\mathcal{X}_{ci} \cap \mathcal{X}_{cj})$ can be computed either by starting with $p(\mathcal{X}_{ci})$ and summing out the variables in $\mathcal{X}_{ci} - \mathcal{X}_{cj}$ or by starting with $p(\mathcal{X}_{cj})$ and summing out variables in $\mathcal{X}_{cj} - \mathcal{X}_{ci}$. The above set of equations ensure that we get the same marginal distribution in either case.

EXAMPLE 8. *Consider a DGP instance $R(A_1, A_2, A_3, A_4)$ with 3 constraints: $|\sigma_{A_1=0 \wedge A_2=0}(R)| = 5$, $|\sigma_{A_2=0 \wedge A_3=0}(R)| = 5$, and $|\sigma_{A_3=0 \wedge A_4=0}(R)| = 5$. Assume $N = 10$ and a binary domain $\{0, 1\}$ for each attribute. The Markov network for this instance is the path graph of Figure 3(a). The maximal cliques are the three edges. We solve the following system of equations to identify the marginals over the edges. The notation $p_{12}(00)$ is a shorthand for $p(X_1 = 0, X_2 = 0)$.*

$$p_{12}(00) + p_{12}(01) + p_{12}(10) + p_{12}(11) = 1 \quad (1)$$
$$p_{23}(00) + p_{23}(01) + p_{23}(10) + p_{23}(11) = 1 \quad (2)$$
$$p_{34}(00) + p_{34}(01) + p_{34}(10) + p_{34}(11) = 1 \quad (3)$$
$$p_{12}(00) = 1/2 \quad (4)$$
$$p_{23}(00) = 1/2 \quad (5)$$
$$p_{34}(00) = 1/2 \quad (6)$$
$$p_{12}(00) + p_{12}(10) = p_{23}(00) + p_{23}(01) \quad (7)$$
$$p_{12}(01) + p_{12}(11) = p_{23}(10) + p_{23}(11) \quad (8)$$
$$p_{23}(00) + p_{23}(10) = p_{34}(00) + p_{34}(01) \quad (9)$$
$$p_{23}(00) + p_{23}(10) = p_{34}(00) + p_{34}(01) \quad (10)$$

Equations 1-3 ensure that the marginals are probability distributions, equations 4-6 ensure that the marginals are consistent with the constraints, and equations 7-10 ensure the marginals produce the same "submarginals" $p(X_2)$ and $p(X_3)$.

We refer to the algorithm of Figure 2 that uses chordal graph method to identify $p(\mathcal{X})$ as CLPALG. For the DGP instance in Example 8, CLPALG solves an LP with 12 variables and we can show that LPALG uses with 16 variables. While this difference is small, for a similar "path-graph" instance with 10 attributes and domain size $D = 10$, we can show that CLPALG uses $9 \cdot 10^2 = 900$ variables while LPALG uses 10^{10} variables.

Algorithm based on Markov Blankets

The second algorithm is structurally similar to the algorithm based on chordal graphs. It solves for a set of low-dimensional marginal distributions using the input constraints and combines these distributions to get a generative probabilistic distribution. We first introduce definitions required to present the algorithm. Let $G = (\mathcal{X}, E)$ denote the Markov network corresponding to the DGP instance.

DEFINITION 2. *The Markov blanket of a set of variables $\mathcal{X}_A \subseteq \mathcal{X}$, denoted $\mathsf{M}(\mathcal{X}_A)$, is defined as: $\mathsf{M}(\mathcal{X}_A) = \{X_i | (X_i, X_j) \in E \wedge (X_i \notin \mathcal{X}_A) \wedge (X_j \in \mathcal{X}_A)\}$.*

The Markov blanket of $\mathcal{X}_A$ is the set of neighbors of vertices in $\mathcal{X}_A$ not contained in $\mathcal{X}_A$. For example, in Figure 3(a), $\mathsf{M}(\{X_2\}) = \{X_1, X_3\}$. We use $\overline{\mathsf{M}}(\mathcal{X}_A)$ as shorthand for $\mathsf{M}(\mathcal{X}_A) \cup \mathcal{X}_A$.

The formal algorithm using Markov blankets is presented in Figure 5. In Step 2, we identify maximal cliques $\mathcal{X}_{c1}, \ldots, \mathcal{X}_{cl}$ in G. In Step 3, we solve for the marginal distributions $p(\overline{\mathsf{M}}(\mathcal{X}_{c1})), \ldots, p(\overline{\mathsf{M}}(\mathcal{X}_{cl}))$. We do this by setting up a system of linear equations quite similar to the one described earlier. In the final step, we construct a generative probability distribution $p(\mathcal{X})$ by combining the marginals identified in the previous step.

Constructing $p(\mathcal{X})$ using marginals $p(\overline{\mathsf{M}}(\mathcal{X}_{ci}))$ is quite involved and uses a canonical representation of probability distributions presented in [1]. We present these details in the full version of the paper. The full version also includes some subtleties relating to ensuring the $p(\mathcal{X})$ is a positive distribution that is required for the above construction to work and describes how to sample from $p(\mathcal{X})$ using Gibbs sampling.

Complexity: We measure the complexity of the algorithm by the number of variables the algorithm creates. This is because theoretically a linear program can be solved in time polynomial in the number of variables. In practice, the actual running time depends on the efficiency of the chosen linear programming solver. For the chordal graph approach, suppose γ is the size of the largest maximal clique of G_c identified in Step 3. Then, the number of the variables in the linear equations is upper bounded by $O(lD^\gamma)$, where l is the number of maximal cliques identified in step 3 and D, the domain size. For the Markov blanket approach, let $\gamma = \max_i |\overline{\mathsf{M}}(\mathcal{X}_{ci})|$. We can show that the number of the variables is upper bounded by $O(lD^\gamma)$. Depending on the actual Markov network, one approach can be more efficient than another. The following example illustrates this claim.

EXAMPLE 9. *Consider the 3×3 grid Markov network G shown in Figure 3(d). The maximal cliques of G are its edges, so the marginals solved by the algorithm correspond to $\overline{\mathsf{M}}(\{X_1, X_2\}) = \{X_1, X_2, X_3, X_4, X_5\}$, $\overline{\mathsf{M}}(\{X_2, X_3\}) = \{X_1, X_2, X_3, X_5, X_6\}$, and so on. More generally, consider an $n \times n$ grid Markov network G. Since each Markov blanket is of a constant size and there are $2(n-1)n$ edges, Markov blankets approach uses at most $2n(n-1) \times |D|^{O(1)}$ variables in its LP. We can show that the size of the largest maximal clique of any chordal super-graph of G is at least $n+1$, and therefore the chordal-graph approach uses $|D|^{n+1}$ variables in its LP, which is less efficient. In contrast, for the Markov networks in Figure 3(a)-(c), we can show that the chordal graph method is more efficient.*

Maximum-Entropy based approaches: Another approach to identifying a generative probability distribution $p(\mathcal{X})$ (in Figure 2) is to pick a distribution with *maximum entropy (MaxEnt)* that satisfies the constraints $C_1, \ldots, C_m$. Identifying such MaxEnt distributions is the subject of recent work on cardinality estimation using query feedback [27,

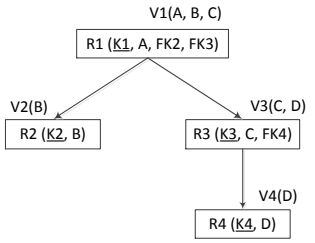

Figure 6: Example snowflake schema

28]. MaxEnt-based approaches have two drawbacks: (1) Current techniques for identifying MaxEnt distributions cannot handle complex constraints involving joins and (duplicate eliminating) projections; (2) Identifying a MaxEnt distribution involves solving a set of non-linear equations and is therefore expensive. In other words, settling for non-MaxEnt distributions allows us to stay within linear programming and also handle a larger class of constraints.

4.3 Multiple Tables

In this section, we present our algorithms for data generation problem involving multiple tables. For the rest of this section, assume a DGP instance involving relations $R_1, \ldots, R_n$ and constraints $C_1, \ldots, C_m$. Each constraint C_j is of the form $|\sigma_{P_j}(R_{i_1} \bowtie \cdots \bowtie R_{i_s})| = k_j$.

We assume that the tables $R_1, \ldots, R_n$ form a *snowflake* schema [11] and all joins are foreign key joins. A snowflake schema has a central *fact table* and several *dimension tables* which form a hierarchy. We can represent a snowflake schema as a rooted tree T with nodes corresponding to the tables $R_1, \ldots, R_n$, and directed edges corresponding to foreign key relationships. The root of the tree is the fact table. Each relation has a single *key* attribute, zero or more *foreign keys* that reference keys of other tables, and any number of non-key attributes that we call *value* attributes. We make the fairly natural assumption that selection predicates in constraints involve only value attributes. We note that keys and foreign keys also represent constraints that need to be satisfied by an output instance. We can extend our algorithms to work with non-snowflake schemas, e.g., directed acyclic graphs instead of trees; however, we do not have algorithms that can handle non-foreign key joins and designing such algorithms is future work.

EXAMPLE 10. *Figure 6 shows four relations R_1, R_2, R_3, R_4 that form a snowflake schema with R_1 being the fact table. The keys of the relations are shown underlined and the foreign keys are named by prefixing "F" to the key that they reference, e.g., FK_2 is the foreign key referencing $R_2.K_2$. The value attributes are A, B, C, D. Two example constraints are $|\sigma_{C=5 \wedge D=2}(R_3 \bowtie R_4)| = 20$ and $|\sigma_{D=2}(R_1 \bowtie R_3 \bowtie R_4)| = 30$.*

We use regular tree terminology to specify relationships between tables; e.g., R_1 is the parent of R_2 in Figure 6.

For each relation R_i, we define a *view* V_i formed by joining all its descendant tables and projecting out non-value attributes. This projection is *duplicate preserving* unlike the projections in our constraints. For example, in Figure 6, $V_3 = \overline{\pi}_{C,D}(R_3 \bowtie R_4)$, where we use $\overline{\pi}$ to denote duplicate preserving projection. With this definition, we can rewrite each constraint C_j as simple selection constraint over exactly one of the views V_i. For instance, the first constraint in Example 10 can be rewritten as $|\sigma_{C=5 \wedge D=2}(V_3)| = 20$.

INPUT: A data generation problem involving $R_1, \ldots, R_n$ and constraints $C_1, \ldots, C_m$.
OUTPUT: Instances of $R_1, \ldots R_n$ satisfying constraints

1. Generate instances of each view V_i satisfying constraints associated with V_i
2. Root to leaf: Update each view V_i:
   $V_i \leftarrow V_i \cup (\pi_{Attr(V_i)}(V_{pi}) - V_i)$
   where V_{pi} is the parent of V_i
3. Generate instances $R_1, \ldots, R_n$ from $V_1, \ldots, V_n$

Figure 7: Algorithm for multiple tables

The above observation forms the basis of our algorithm, which is presented in Figure 7. In Step 1, we generate an instance of each view V_i that satisfies all cardinality constraints associated with it. Since the constraints are all "single table" selection constraints, we can use algorithms from Section 4.2 to generate these instances. However, these independently generated view instances may not correspond to valid relation instances: Consider relation instances $R_1, \ldots, R_n$ that satisfy all key-foreign key constraints and let R_{p_i} denote the parent of some relation R_i. We can show that views V_i and V_{p_i} should satisfy the property $\pi_{Attr(V_i)}(V_{pi}) \subseteq V_i$ (note: π is duplicate eliminating). For example, in Figure 6, every distinct value of B in $V_1(A, B, C)$ occurs in some tuple $V_2(B)$. However, the view instances generated in Step 1 may not satisfy this property. Therefore, in Step 2, we add additional tuples to each V_i to ensure that this containment property is satisfied in the resulting view instances. These updates might cause some cardinality constraints to be violated, however we show shortly how the degree of these violations can be bounded. In the final step, we construct relation instances $R_1, \ldots, R_n$ consistent with $V_1, \ldots, V_n$.

We now discuss how to minimize the error introduced in Step 2. (Error is the absolute difference between required and actual cardinalities of expressions in constraints.) Recall that all of our algorithms use the intervalization trick (see Section 4.1). With intervalization, the value of an attribute is constrained to come from some interval $[v_i, v_j)$ during data generation, but we are free to select any value from the interval. We can minimize the error introduced in Step 2 by picking values from an interval in a consistent manner across all views. The following example illustrates this idea.

EXAMPLE 11. *Consider two relations $R_1(K_1, A, FK_2)$ and $R_2(K_2, B)$ and four constraints:*

$$|\sigma_{B \in [1,5)}(R_2)| = 2 \qquad |\sigma_{B \in [1,2)}(R_1 \bowtie R_2)| = 2$$
$$|\sigma_{B \in [5,10)}(R_2)| = 3 \qquad |\sigma_{B \in [2,10)}(R_1 \bowtie R_2)| = 2$$

Assume domain size $D = 10$. To identify view instances $V_1(A, B)$ and $V_2(B)$ that satisfy all cardinality constraints, LPALG *solves the following two LPs (using intervalization):*

$$x_{[1,10)[1,2)} = 2 \qquad y_{[1,2)} + y_{[2,5)} = 2$$
$$x_{[1,10)[2,5)} + x_{[1,10)[5,10)} = 2 \qquad y_{[5,10)} = 3$$

Here $x_{[\ldots)}$ denote LP variables corresponding to V_1 and $y_{[\ldots)}$ denote LP variables corresponding to V_2. One solution of these LPs is $x_{[1,10)[1,2)} = 2$, $x_{[1,10)[2,5)} = 1$, $x_{[1,10)[5,10)} = 1$,

$y_{[1,2)} = 0$, $y_{[2,5)} = 2$, $y_{[5,10)} = 3$. *If we pick the minimum possible value when selecting attribute values from an interval, we get the instances of* V_1 *and* V_2 *(without boxed* 1*) shown below.*

A	B
1	1
1	1
1	2
1	5

$V_1(A, B)$

B
2
2
5
5
5
1

$V_2(B)$

K_1	A	FK_2
1	1	6
2	1	6
3	1	1
4	1	4

$R_1(K_1, A, FK_2)$

K_2	B
1	2
2	2
3	5
4	5
5	5
6	1

$R_2(K_2, B)$

In Step 2 of our algorithm, we add the (boxed) tuple 1 *to* $V_2(B)$ *to ensure* $\pi_B(V_1) \subseteq V_2$. *This update results in an additive error of* 1 *in one of the constraints. The instances of* R_1 *and* R_2 *that are consistent with* V_1 *and* V_2 *are also shown above. We can verify that randomly and independently picking values from intervals results in a solution with a larger error*[7].

In the full version, we present rounding techniques that together with the intervalization trick above ensure that the *additive error* of any constraint is bounded by $O(m)$, the number of cardinality constraints. The full version also presents extensions to the algorithm in Figure 7 that can handle *degree constraints*, e.g., each tuple in R_2 is referenced by at most 10 tuples in R_1. Such constraints allow us to have finer control over the joins across relations.

4.4 Projections

This section briefly discusses how to handle cardinality constraints with projections. Recall from Section 2 that only duplicate eliminating projections are useful as constraints. The general data generation problem involving projections is very hard as the NEXP-completeness result suggests. In the full version, we show that the data generation problem over a single table with constraints that have just projections and no selections is nontrivial and has connections to known hard problems in combinatorial geometry.

Here we present an algorithm for the case of a single table and a single attribute. The ideas behind this algorithm can be combined with techniques from earlier sections to derive a more general solution for the case where attributes involved in different projection constraints are non-overlapping, meaning, if $|\pi_{\mathcal{A}_1}(\ldots)| = k_1$ and $|\pi_{\mathcal{A}_2}(\ldots)| = k_2$ are two constraints then either $\mathcal{A}_1 = \mathcal{A}_2$ or $\mathcal{A}_1 \cap \mathcal{A}_2 = \emptyset$.

Let $R(A)$ denote the table being generated. Each constraint C_j has one of two forms: $|\pi_A(\sigma_{A\in[l_j,h_j)}(R))| = k_j$ or $|\sigma_{A\in[l_j,h_j)}(R)| = k_j$. We identify basic intervals $[v_i, v_{i+1})$ $(1 \le i < l)$ exactly as we did in Section 4.1. We now introduce two variables $x_{[v_i,v_{i+1})}$ and $y_{[v_i,v_{i+1})}$ for each basic interval; $x_{[v_i,v_{i+1})}$ denotes the number of tuples of $R(A)$ belonging to the interval and $y_{[v_i,v_{i+1})}$, the number of distinct tuples belonging to the interval. We generate one linear equation corresponding to each constraint using x_* variables for constraints not involving a projection and y_* variables for constraints involving projections. For each interval $[v_i, v_{i+1})$, we add two additional equations: $y_{[v_i,v_{i+1})} \le x_{[v_i,v_{i+1})}$ and $y_{[v_i,v_{i+1})} \le (v_{i+1} - v_i)$. The first equation captures the constraint that in any interval the number of distinct values is not greater than the number of values (counting duplicates) and the second captures the natural bound on the number of distinct values in the interval.

As in Section 4.2, we solve the LP and perform randomized rounding. One tricky case happens when $q \le y_{[v_i,v_{i+1})} \le x_{[v_i,v_{i+1})} < (q+1)$ where q is some nonnegative integer. If we independently round $x_{[v_i,v_{i+1})}$ and $y_{[v_i,v_{i+1})}$, we might end up with $y = q+1$ and $x = q$, which is inconsistent. To resolve this problem, we use a slightly different rounding procedure: We pick a random value r uniformly between q and $q+1$. We round $x_{[v_i,v_{i+1})}$ (resp., $y_{[v_i,v_{i+1})}$) to $q+1$ if $x_{[v_i,v_{i+1})} > r$ (resp., $y_{[v_i,v_{i+1})} > r$) and to q, otherwise. It is not hard to see that the solution is consistent and satisfies all constraints in expectation.

5. EXPERIMENTS

5.1 Setup

To generate a large number of meaningful cardinality constraints, we consider the following hypothetical scenario: We have an instance of TPC-H benchmark database and our goal is to generate a synthetic database instance such that a workload of 8 queries Q1-Q10 (not including Q4 and Q9) has similar performance characteristics over the synthetic and original database. We do not consider queries Q4 and Q9 since Q4 has a nonequality predicate between two attributes and Query Q9 has a LIKE predicate as the main predicate of the query, and our algorithms currently do not handle either of these types of predicates. Each TPC-H query has an associated parameterization, and our workload consists of various parameterizations of these 8 queries. For example, in query Q3, we can substitute the parameter `[SEGMENT]` with five different values and the parameter `[DATE]` with 30 different values.

The value distributions in TPC-H are fairly simple, and we can "cheat" and generate an instance similar to the original TPC-H database by generating attribute values almost independently. But such generation would not be possible if our database had the TPC-H schema, but complex value correlations. We note in this context that once we fix the constraints, the performance of our algorithms is not sensitive to such correlations—these correlations only change the cardinalities of the constraints, not the structure of the constraints and the performance of our algorithms depends on the structure of the constraints, not cardinality values.

If we assume that the performance of queries is a function of various intermediate join cardinalities, we get the following methodology for generating constraints: For each query, we consider the query plan produced by the query optimizer and the various nodes in the query plan. Each node corresponds to a relational expression, and we identify all nodes whose relational expressions involve only joins and selections; we evaluate all such relational expressions for various parameterizations, and use the resulting expressions and cardinalities as input constraints. This produces a set of 1100 constraints ranging from single table constraints to join constraints involving all the TPC-H tables.

In the following, we report on various performance characteristics of our algorithms when run over various subsets of these 1100 constraints. Although we do not explicitly present this result, the synthetic database generated using

[7]There exist techniques for consistently picking attribute values from intervals that ensure greater randomness in the output instances compared to the strategy of picking the minimum.

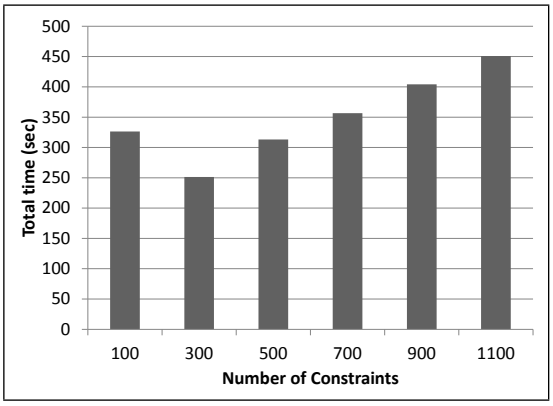

Figure 8: Total runtime

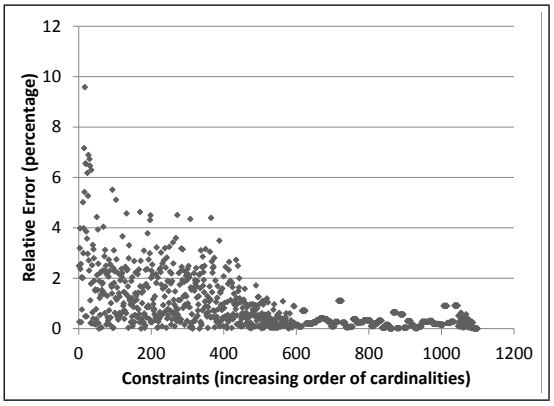

Figure 9: Relative errors in constraints

all of these 1100 constraints as input does indeed have a performance characteristic similar to the original database for the workload of 8 queries we start with.

Our overall experiment methodology is very similar to the methodology used by [7, 22]. Also, since QAGen [6], reports performance numbers over TPC-H as well, we can (roughly) compare the performance of our algorithm with that of QAGen using the numbers reported in [6].

All of our experiments were run on a dual core 2.4 GHz machine with 6GB of main memory.

5.2 Results

We now report various performance results of our algorithms. In all of the results that we report here, we use the CLPAlg algorithm for identifying probability distribution within a single view. The algorithm that uses markov blankets has a worse performance than CLPAlg for this class of inputs. For solving the the linear equations, we used a commercial state-of-the-art LP solver. Unless mentioned otherwise, we use TPC-H 1G for all of our experiments.

Figure 8 shows the overall running time of our algorithms as we vary the number of input constraints. To vary the number of constraints, we first ordered all the constraints by the queries from which they were obtained. For example, the constraints from Q1 occurred before constraints from Q2 in this ordering, and we picked various prefixes of this ordering to obtained subsets of different sizes.

Recall that our overall algorithm has three stages: In the first stage, the algorithm sets up a linear program by analyzing the input constraints, in the second step, the linear program is solved using the LP solver. The solution of the linear program represents a probability distribution for each view, and in the final step our algorithm samples from this probability distribution to produce the output database. Most of the overall time of our algorithm goes into the third and final stage. The first stage takes negligible amount of time and we report the time taken for the second stage (LP solving) shortly. Overall we note that our algorithm is able to generate a database instance for an input involving all the 1100 constraints in less than 10 minutes.

Figure 9 shows the relative error of all the 1100 constraints (when we use all the 1100 constraints as input). Each constraint is shown as a point in Figure 9. To better highlight the errors, we ordered all the constraints (on the horizontal axis) based on their cardinalities; constraints with larger cardinalities are shown to the right and constraints with smaller cardinalities are shown to the left. The relative error is shown on the vertical axis. As expected (and predicted by standard sampling theory), the constraints with larger cardinalities have smaller relative error and constraints with smaller cardinalities have a larger relative error. But overall, most of the constraints are satisfied with less than 5% error.

Table	*Extra tuples*	*Table size*
LINEITEM	0	6000000
ORDERS	24	1500000
CUSTOMER	18	150000
PARTSUPP	20	800000
PART	202	200000
SUPPLIER	0	10000

Figure 10: Error due to additional referential integrity tuples

There are two sources of errors in our algorithm. The first is introduced by sampling, and the second due to addition of extra tuples in Step 2 of Figure 6 to ensure referential integrity. Figure 10 shows the extra tuples added for each of the 6 tables (with foreign keys) in TPC-H schema and the overall table size. Figure 10 suggests that the error introduced in this step is very small.

Our next set of experiments cover the size and complexity of the linear program generated by our algorithm, and the overall time taken to solve the linear program. Figure 11 shows the number of variables introduced by our algorithm as we increase the number of constraints. There exists a sudden jump in the number of constraints—this jump occurs due to addition of constraints from Q8, which is a large join query involving all the tables of the TPC-H schema. Figure 12 shows the number of linear equations introduced by our algorithm and Figure 13 shows the overall size of the linear program setup by our algorithm. These two figures also show a similar jump when query Q8 is introduced.

Figure 14 shows the time required by the LP solver to solve the generated linear program as a function of the number of constraints. The time required to solve the LP is less than 5 seconds even if we input all of the 1100 constraints. As mentioned in Section 1, an important aspect of our algorithm is that it is not very sensitive to the size of the database being generated. Figure 15 shows the time to solve the LP for constraints derived from TPCH 0.1G database, and we note

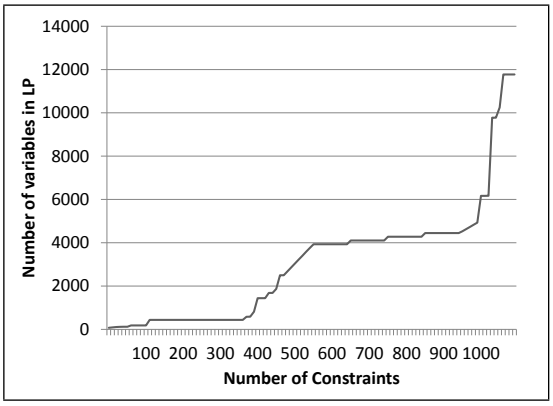

Figure 11: Number of variables in the LP

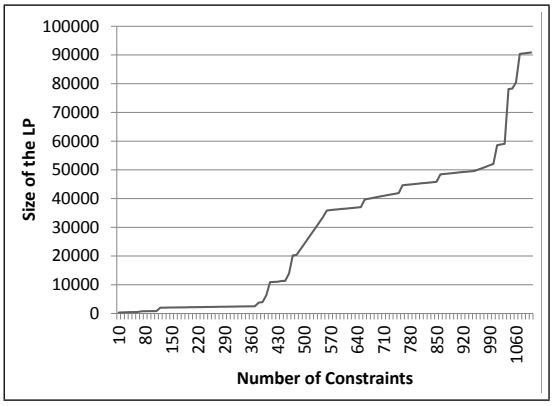

Figure 13: Size of the LP

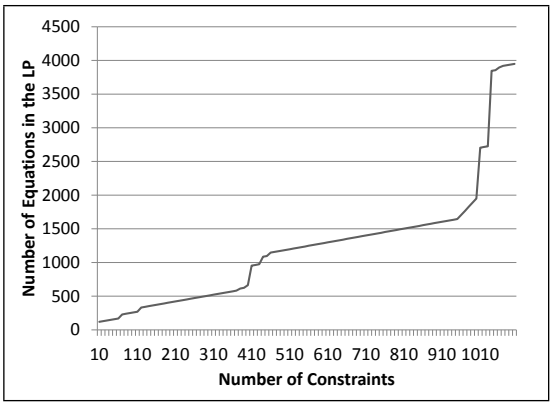

Figure 12: Number of equations in the LP

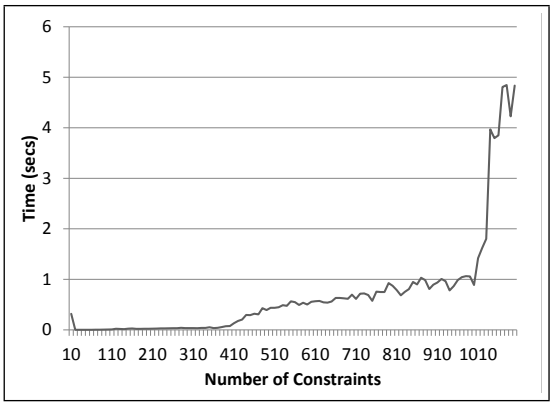

Figure 14: Time to solve the LP

that there is very little difference when we move from 0.1G to 1G.

6. RELATED WORK

There is much work on generating synthetic databases [7, 6, 15, 22, 24, 5]. Current database generation tools [15, 18, 7] allow users to specify the data distributions of individual attributes or somewhat simplistic intra-attribute correlations. For example, DGL (Data Generation Language) [7] can specify something like "produces 65% of uniformly distributed and 35% of normally distributed rows". Houkjaer et al. [17] proposed to use a graph model, which holds the primary-foreign keys and various other statistical information, to guide the data generation process. While being very efficient in producing large database instances, those tools do not have much control over more complex data characteristics, especially over the input/output sizes of the queries, which are particularly important in testing the performance of query plans or optimizers. Mannila et al. [24] first studied query-aware data generation. However, their focus is to generate a small yet complete test dataset for the queries. Bruno et al. [9] considered the problem of generating query instances, rather than generating database instances, that satisfy cardinality constraints on their sub-expressions.

There is also much work on generating non-relational datasets. Several XML generators have been developed, e.g., the Winsconsin XML generator [2], ToxGene [4] and GxBE [12]. Among those, GxBE [12] is perhaps the closest to ours in spirit in that it also allows users to specify cardinality constraints over XPath queries. Olston et al. [25] studied the problem of generating example data for dataflow programs (e.g., MapReduce and Pig Latin). However, their goal is to produce a small dataset to illustrate the operator behaviors as well as helping human understanding the program. There is also a huge literature on generating graph data (see e.g., [3, 21, 32, 20]). Typically, their goals are to generate random graphs with certain properties on degree distribution, edge density, diameter, and so on.

Recently, there have been increasing interests in data masking and privacy preserving techniques (see e.g., [29, 33, 23, 13, 10] and references therein). However, the fundamental difference of most of those work from ours is that they generate a new database instance by modifying an existing one, while our data generation algorithms take only the constraints as input (even though the constraints may be extracted from existing databases).

7. CONCLUSION

We considered the problem of generating synthetic databases and proposed a declarative approach for specifying data characteristics using cardinality constraints. We showed the problem is extremely hard in general, and developed a suite of efficient exact or approximate algorithms for many important classes of constraints. Our experimental results have demonstrated the effectiveness and efficiency of our approach.

Our work opens up many avenues for further research. For instance, it would be very interesting and important

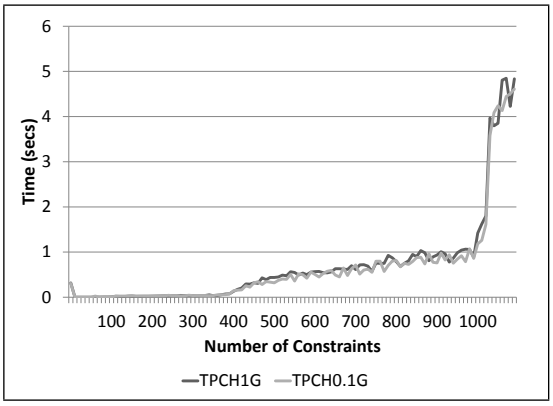

Figure 15: Time to solve the LP (1G vs 0.1G)

if one can handle cardinality constraints with overlapping projections. We also intent to extend our current algorithms to handle more general schemas and joins.

8. REFERENCES

[1] P. Abbeel, D. Koller, and A. Y. Ng. Learning factor graphs in polynomial time and sample complexity. *J. of Machine Learning Research*, 7:1743–1788, 2006.

[2] A. Aboulnaga, J. F. Naughton, and C. Zhang. Generating synthetic complex-structured XML data. In *WebDB*, pages 79–84, 2001.

[3] W. Aiello, F. Chung, and L. Lu. A random graph model for power law graphs. *Experimental Mathematics*, 10(1):53–66, 2001.

[4] D. Barbosa, A. O. Mendelzon, J. Keenleyside, et al. ToXgene: An extensible template-based data generator for XML. In *WebDB*, pages 49–54, 2002.

[5] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *ICDE*, pages 506–515, 2007.

[6] C. Binnig, D. Kossmann, E. Lo, et al. QAGen: generating query-aware test databases. In *SIGMOD*, pages 341–352, 2007.

[7] N. Bruno and S. Chaudhuri. Flexible database generators. In *VLDB*, pages 1097–1107, 2005.

[8] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A multidimensional workload-aware histogram. In *SIGMOD*, pages 211–222, 2001.

[9] N. Bruno, S. Chaudhuri, and D. Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE Trans. Knowl. Data Eng.*, 18(12):1271–1275, 2006.

[10] M. Castellanos, B. Zhang, I. Jimenez, et al. Data desensitization of customer data for use in optimizer performance experiments. In *ICDE*, pages 1081–1092, 2010.

[11] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.

[12] S. Cohen. Generating XML structure using examples and constraints. *PVLDB*, 1(1):490–501, 2008.

[13] C. Dwork. Differential privacy. In *ICALP (2)*, pages 1–12, 2006.

[14] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *SIGMOD*, pages 461–472, 2001.

[15] J. Gray, P. Sundaresan, S. Englert, et al. Quickly generating billion-record synthetic databases. In *SIGMOD*, pages 243–252, 1994.

[16] J. M. Hammersley and P. Clifford. Markov fields on finite graphs and lattices. Unpublished manuscript.

[17] K. Houkjaer, K. Torp, and R. Wind. Simple and realistic data generation. In *VLDB*, pages 1243–1246, 2006.

[18] IBM DB2 test data generator. `http://www.ibm.com/developerworks/data/library/techarticle/dm-0706salkosuo/index.html`.

[19] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer Verlag, 2005.

[20] S. Lattanzi and D. Sivakumar. Affiliation networks. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 427–434. ACM, 2009.

[21] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, et al. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *PKDD*, pages 133–145, 2005.

[22] E. Lo, N. Cheng, and W.-K. Hon. Generating databases for query workloads. In *VLDB*, pages 848–859, 2010.

[23] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):3, 2007.

[24] H. Mannila and K.-J. Raiha. Automatic generation of test data for relational queries. *J. Comp. Syst. Sci*, 38(2):240–258, 1989.

[25] C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *SIGMOD*, pages 245–256, 2009.

[26] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[27] C. Re and D. Suciu. Understanding cardinality estimation using entropy maximization. In *PODS*, pages 53–64, 2010.

[28] U. Srivastava, P. J. Haas, V. Markl, et al. ISOMER: consistent histogram construction using query feedback. In *ICDE*, 2006.

[29] L. Sweeney. k-Anonymity: A Model for Protecting Privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst*, 10(5):557–570, 2002.

[30] T. Syrjanen. *Logic Programs and Cardinality Constraints: Theory and Practice*. PhD thesis, Helsinki University of Technology, 2009.

[31] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. of Comput*, 13(3):566–579, 1984.

[32] J. Winick and S. Jamin. Inet-3.0: Internet topology generator, 2002. Technical Report CSE-TR-456-02, University of Michigan, Ann Arbor.

[33] W. E. Winkler. Masking and re-identification methods for public-use microdata: Overview and research problems. In *Privacy in Statistical Databases*, pages 231–246, 2004.