Flexible Database Generators

Nicolas Bruno Surajit Chaudhuri

Microsoft Corp.
One Microsoft Way, Redmond, WA 98052
{nicolasb,surajitc}@microsoft.com

Abstract

Evaluation and applicability of many database techniques, ranging from access methods, histograms, and optimization strategies to data normalization and mining, crucially depend on their ability to cope with varying data distributions in a robust way. However, comprehensive real data is often hard to come by, and there is no flexible data generation framework capable of modelling varying rich data distributions. This has led individual researchers to develop their own ad-hoc data generators for specific tasks. As a consequence, the resulting data distributions and query workloads are often hard to reproduce, analyze, and modify, thus preventing their wider usage. In this paper we present a flexible, easy to use, and scalable framework for database generation. We then discuss how to map several proposed synthetic distributions to our framework and report preliminary results.

1 Introduction

When designing or modifying a new DBMS component or technique, it is crucial to evaluate its effectiveness for a wide range of input data distributions and workloads. Such a systematic evaluation helps identify design problems, validate hypothesis, and evaluate the robustness and quality of the proposed approach.

Consider for instance a new histogram technique for cardinality estimation (some recently proposed approaches include [7, 10, 14, 15]). Since these techniques often use heuristics to place buckets, it is very difficult to study them analytically. Instead, a common practice to evaluate a new histogram is to analyze its efficiency and quality of approximations with respect to a set of data distributions. For such a validation to be meaningful, input data sets must be carefully chosen to exhibit a wide range of patterns and characteristics. In the case of multidimensional histograms, it is crucial to use data sets that display varying degrees of column correlation, and also different levels of skew in the number of duplicates per distinct value. Note that histograms are not just used to estimate the cardinality of range queries, but also to approximate the result size of complex queries that might involve joins and aggregations. Therefore, a thorough validation of a new histogram technique needs to include data distributions with correlations that span over multiple tables (e.g., correlation between columns in different tables connected via foreign key joins).

Consider the problem of automatic physical design for database systems (e.g., [3, 9, 18]). Algorithms addressing this problem are rather complex and their recommendations crucially depend on the input databases. It is therefore advisable to check whether the expected behavior of a new approach (both in terms of scalability and quality of recommendations) is met for a wide variety of scenarios. For that purpose, test cases should not be simplistic, but instead exhibit complex intra- and inter-table correlations. As an example, consider the popular TPC-H benchmark. While TPC-H has a rich schema and syntactically complex workloads, the resulting data is mostly uniform and independent. In the context of physical database design, we can ask ourselves how would recommendations change if the data distribution shows different characteristics. What if the number of `lineitems` per order follows a Zipfian distribution? What if `customers` buy `lineitems` that are supplied exclusively by vendors in their `nation`? What if `customer` balances depend on the total price of their respective open `orders`? These constraints require capturing dependencies across tables.

In many situations, synthetically generated databases are the only choice: real data might not be available at all, or it might not be comprehensive enough to thoroughly evaluate the system in consideration. Unfortunately, to our knowledge there is no available data

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

generation framework in the context of RDBMSs that is capable of modelling varying rich data distributions like the examples described above (references [2, 4] discuss some recent frameworks in the context of XML databases). This has led individual researchers to develop their own ad-hoc generation tools. Usually, generation settings are not clearly documented and details become hidden in the particular implementations. As a consequence, resulting data distributions and query workloads are often hard to reproduce, analyze, and modify, which severely limits their applicability.

In this paper we present a flexible framework to specify and generate databases that can model data distributions with rich intra- and inter-table correlations. In Section 2 we introduce a simple special purpose language with a functional flavor, called DGL (Data Generation Language). DGL uses the concept of *iterators* as basic units that can be composed to produce streams of tuples. DGL can also interact with an underlying RDBMS and leverage its well-tuned and scalable algorithms (such as sorting, joins, and aggregates). As we show in Section 3, DGL can also be used to further simplify the specification of synthetic relational databases by adding "DGL annotations" to the SQL CREATE TABLE statement, which concisely specify how a table should be populated. Finally, Section 4 reports some preliminary results on the data generation framework introduced in this paper.

2 Data Generation Language (DGL)

We next present DGL, a special purpose language to generate synthetic data distributions.

2.1 Data Types

We now discuss the main data types in DGL, and the various operations that each data type support.

Scalars are the most basic type in DGL, and are further subdivided into integers (`Int`), double precision reals (`Double`), strings (`String`), and dates (`DateTime`). DGL supports the traditional operations over each scalar type, such as arithmetic and comparison operations for integers and real numbers, and concatenation and substring operations for strings.

Rows are fixed-size heterogenous sequences of scalars. For instance, `R = [1.0, 2, 'John']` has type `[Double, Int, String]`. The expression `dim(R)` returns the number of columns in `R`, and `R[i]` returns the *i-th* column of `R` (`0≤i<dim(R)`). In the example above, `dim(R)=3`, `R[0]=1.0`, and `R[0]+R[1]=3.0`. The operator `++` combines rows, so `[2,'John'] ++ ['Smith',4]` returns a new row `[2,'John','Smith',4]`. Finally, operators defined over scalars can use rows as operands provided that (i) each operand is of the same size, and (ii) each column in the operands is compatible with the operator. In that case, the result is a new row where each column is obtained by applying the operator to the individual operands' columns. Then, `[1,2,3]+[4,5,6] = [5,7,9]`, and both `[1,2]+[2,3,4]` and `[1,'John']+[3,4]` are invalid expressions.

Iterators are the most important objects in DGL. Iterators support the operations `open` (which initializes the iterator), and `getNext` (which returns a new `Row` or an end-of-iterator special value). For instance, the iterator `Step(f,t,s)` results in the following rows {`[f]`, `[f+s]`, `[f+2s]`, ..., `[f+ks]`} where `f+ks` ≤ `t` < `f+(k+1)s`. As another example, for a row `R`, the iterator `Constant(R)` returns the infinite sequence {`R, R, R, ...`}. All operations on rows can also be applied to iterators. The result of such operations is a new iterator that, for each `getNext` request, obtains a new row from each operand, applies the operator to such rows, and returns the resulting row. Consider `I1=Step(1,100,2)`, `I2=Step(5, 21, 3)`, and `I3=Constant([10,20,30])`. In that case,

```
      I3[1] =  {[20], [20],...}
    I1++I2 =  {[1,5], [3,8], ..., [11,20]}
I1+I2-I3[0] =  {[-4], [1], ..., [21]}
```

Tables are in-memory instances of (finite) iterators provided for efficiency purposes. We denote the *i-th* row in table `T` as `T[i]`.

Implicit casts

DGL applies implicit casts to operands depending on the expression context. Consider `Step(1,100,1) ++ [5]`. The left operand is an iterator and the right operand is a row. In this case, the row is implicitly promoted to iterator `Constant([5])`. Similarly, in `[1]+3`, the integer `3` is converted into a single-column row `[3]`. Iterators and tables can be used interchangeably (but tables must fit in memory). We rely on implicit casts in the remainder of this paper.

2.2 Primitive Iterators

If `Step` and `Constant` were the only available iterators in DGL there would not be many interesting data sets that we could generate. A strength of DGL is the existence of a large set of primitive iterators and the ability to extend this basic set with new user-defined primitives. We next describe some built-in iterators in DGL.

Distributions. DGL natively supports many statistical distributions as built-in iterators. For instance, function `Uniform` takes two input iterators and returns an iterator that produces a multidimensional uniform distribution. Consider

I=Uniform(iLO, iHI). Each time a new row is required from this iterator, I retrieves the next row from iterators iLO and iHI (denoted rLO and rHI) and produces a new row (denoted rOut) where rOut[i] is a random number between rLO[i] and rHI[i]. Then, Uniform(Constant([5,5]), Constant([8,15])) is an iterator that produces an infinite stream of uniformly distributed 2-dimensional points (x,y) where $5 \leq x \leq 8$ and $5 \leq y \leq 15$. Due to implicit casts, we can write the same iterator as Uniform([5,5], [8,15]). In addition to Uniform, DGL supports a wide variety of discrete and continuous iterators to produce intra-table correlated distributions, including UniformInt (discrete version of Uniform), Normal, Exponential, Zipfian, and Poisson[1].

SQL and Relational Queries. Two primitive functions, Persist and Query, are provided to bridge DGL and an underlying RDBMS. Persist takes an iterator I and an optional string s as its input, bulk-loads all the rows provided by I in a database table named s, and returns the string s. If no input string is provided, DGL generates a unique name for a temporary table in the DBMS. In this case, the bulk-loaded table contains one additional column, id, that stores the sequential number of the corresponding row. The remaining columns in the table are called v0, v1, and so on. For instance, if Persist(Step(15,1000,6)++5) returns #Tmp1, a new temporary table #Tmp1 was created in the DBMS as a side effect with the following schema and values:

id	v0	v1
0	15	5
1	21	5
...	...	...
164	999	5

Conversely, Query takes a parametrized query string sqlStr and a sequence of iterators T_0, ..., T_{k-1}, and (i) materializes the *i-th* iterator in the database (using an implicit Persist), (ii) replaces the *i-th* parameter in the query with the temporary table assigned to the corresponding iterator, (iii) executes the resulting query, and (iv) returns an iterator over the query results[2]. The *i-th* parameter is denoted as <i> in sqlStr. The expression below returns a random permutation of all odd numbers below 1000:

```
Query( "SELECT v0 FROM <0> ORDER BY v1",
       Step(1, 1000, 2) ++ Uniform(0, 1) )
```

In this expression, Query takes an iterator of two columns, where the first column consists of all odd numbers between 1 and 1000 and the second is a random number. This iterator is persisted in the database into a temporary table, and is subsequently read in order of the random column v1. This example shows an early decision while designing DGL: instead of reinventing robust and scalable algorithms to sort, aggregate, or join rows, we reuse the underlying DBMS. While, strictly speaking, the loading/query cycle might incur some overhead, we found out that it is quite acceptable while providing much additional functionality and robustness.

Non-blocking duplicate elimination. DGL provides a primitive iterator, dupFilter, which takes as input an iterator I, two numbers f1 and f2, and a row of indexes cols, and controls the degree of uniqueness of I. Each time we call *getNext* on dupFilter(I, f1, f2, cols), we get the next row R from the input iterator I and return it a fraction f1 of the time if the columns of R specified by cols were not seen previously, or a fraction f2 of the time otherwise (f1+f2 is not necessarily equal to one). The pseudo-code of dupFilter.getNext is given in Figure 1. This operator is useful in the context of infinite streams, since it is not possible to first persist them into a temporary table and then use SQL to filter duplicates. DGL supports two implementations of dupFilter which vary on how line 4 in Figure 1 is implemented and balance space consumption and efficiency: dupFilterMem maintains a hash table of all unique values in memory, and dupFilterDB creates a temporary table on the database system with a unique constraint and uses it to check whether a new value was already inserted.

Miscellaneous. DGL supports a large library of primitive iterators, including Top(I,k), which produces only the first k rows of iterator I, Union(I_1, I_2), which produces all rows from iterator I_1 followed by those from iterator I_2, ProbUnion(I_1,I_2,p), which is similar to Union(I_1, I_2) but interleaves rows from I_1 and I_2 with probability p and 1-p, respectively, tableApply(T,I) which returns, at each step, the element in table T indexed by the next element from the integer-iterator I, and Duplicate(IR,IF) which returns, for each r and f respectively retrieved from iterators IR and IF, f copies of r. For instance, Duplicate(Step(5, 8, 1), [3]) returns {[5], [5], [5], [6], [6], [6], [7], [7], [7]}.

[1]With appropriate wrappers, it is also possible to use statistical distributions from other packages such as SAS, MATLAB, or Octave. Those packages, however, can generate distributions with just intra-table correlations.

[2]If the underlying database system has native support for streams, operator Query first creates a user defined stream in the database system, and then directly works over the resulting stream, without necessarily materializing intermediate results.

```
dupFilter.getNext ( local I:Iterator, f1, f2:Double, [c1, ..., cn]:  Row )
01 while(true)
02   R = I.getNext()
03   if (R = end-of-iterator) return R
04   isD = [R[c1], ..., R[cn]] is duplicate
05   if ( isD and random()<f1) or (!isD and random()<f2)
06         return R
```

Figure 1: Pseudo-code for `dupFilter.getNext()`.

2.3 Expressions and Functions

The general form of a DGL expression is as follows:

```
LET   v1 = expr1,
      v2 = expr2,
      ...,
      vn-1 = exprn-1
IN    exprn
```

where each $\texttt{expr}_i$ is a valid DGL expression that can refer to variables $\texttt{v}_j$. The only restriction is that the reference graph be acyclic (primitive and user-defined iterators can be recursive). For instance, the following expression is an iterator that produces 65% of uniformly distributed and 35% of normally-distributed rows (Figure 2 shows graphically the output distribution of this iterator):

```
LET  count = 10000, P = 0.65,
     U = Uniform([5,7], [15,13]),
     N = Normal([5,5], [1,2])
IN   Top( ProbUnion(U, N, P), count )
```

DGL functions, in turn, are specified as follows:

```
function_name (arg1, ..., argn) = expr
```

where `expr` is a DGL expression. For instance, we can define a function `simpleF` that parametrizes the previous example as follows:

```
simpleF (P, count) =
   LET U = Uniform([5,7], [15,13]),
       N = Normal([5,5], [1,2])
   IN  Top(ProbUnion(U, N, P), count)
```

and obtain the original expression using `simpleF(0.65, 10000)`.

2.4 DGL Programs and Evaluation Model

A DGL program is a set of function definitions followed by an expression (called the *main* expression). Evaluating a DGL program is equivalent to evaluating its main expression, casting the result to an iterator, and returning all the rows produced by this iterator. This stream of rows can then be either saved to a file, or discarded if the program already *persisted* tables in the RDBMS as a side effect. We now discuss the evaluation model of a DGL program.

In general, we can see a DGL program as a directed acyclic graph, or *DAG* (see Figure 3 for some examples). The DAG of a general expression consists of

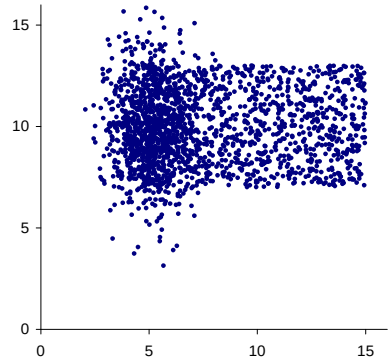

Figure 2: Distribution obtained by a DGL iterator.

one node (with label $\texttt{v}_i$) for each $\texttt{v}_i$=$\texttt{expr}_i$ in the main expression plus an additional node (with label `main`) for the `IN` expression $\texttt{expr}_n$. There is a directed edge between nodes n_i and n_j if $\texttt{expr}_j$ directly refers to $\texttt{v}_i$. If (n_i, n_j) is a directed edge in the DAG, we say that n_j is a *consumer* of n_i.

In the simplest case, if each node has exactly one consumer (e.g., Figure 3(a)), the evaluation of a DGL program is straightforward. Each node produces objects that are passed to its unique consumer. Multiple iterators can be chained in this way and the memory required to evaluate a program remains constant[3]. This evaluation model is still valid even if some nodes have multiple consumers, provided that such nodes are scalars or rows (e.g., node `a` in Figure 3(b) is shared by nodes `b` and `c`). The reason is that scalars and rows are immutable and can be safely shared among consumers.

In the general case, if some iterator node has multiple consumers (e.g., node `b` in Figure 3(c)) the situation is more complex. The problem is that the different consumers must see the same stream of rows from the shared iterator but might request rows at different rates. DGL implements a buffering mechanism for that purpose. It is important to note that, in many common examples (see Section 4) the actual speed of different consumers is the same, and therefore the buffer is always of a small constant size. Consider the example in Figure 3(c). Each time `main` produces a new row, it requests the next row to both `b` and `c`. In this case, `b` is shared between `main` and `c`, so it must buffer the row that produces for `main` until `c` requests it. However, for `c` to generate the row required by `main`

[3]Iterators with an internal state that can grow unbounded (e.g., dupFilterMem) are not be memory-bounded.

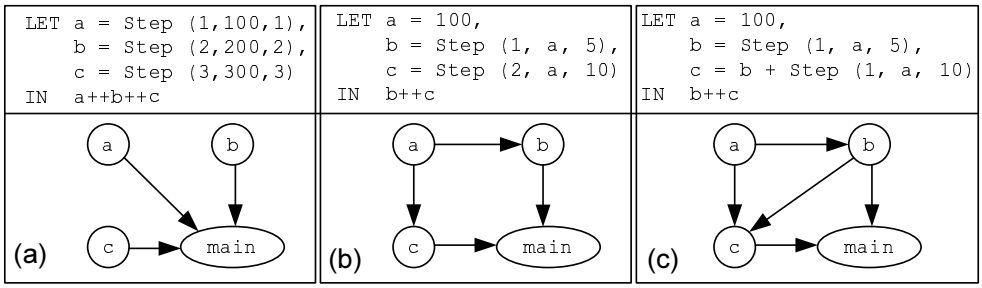

Figure 3: Simple DGL programs and their corresponding DAGs.

it requests the next row of b *in sync* with main. Therefore, although buffering is present in this example, it is not used beyond its first cell and the memory of the whole program remains constant.

Memory/Performance Trade-offs

DGL offers a simple mechanism to allow a program to trade-off space consumption for performance in the presence of shared iterators. Consider the following program:

```
LET  U = Step(1, 10000, 1),
     Q = Query("some complex query", U)
IN   U ++ Q
```

whose DAG is shown in Figure 4(a). Due to the implicit Persist operator generated by Q, iterator U is consumed entirely when the first row from Q is requested. Therefore, U must buffer its entire stream to satisfy future requests from main. In this particular example, U simply produces integers of increasing value. Therefore, a simple way to avoid the buffering of U is to *duplicate* U such that Q and main request rows from different instances. This can be specified as follows (the corresponding DAG is shown in Figure 4(b)):

```
LET  U = Step(1, 10000, 1),
     U'= Step(1, 10000, 1),
     Q = Query("some complex query", U)
IN   U' ++ Q
```

In this case each iterator node has at most one consumer and there is no buffering involved. Since the processing done by U is minimal, there is almost no additional overhead. Unfortunately, in complex programs it is not always desirable to duplicate iterators, since each copy duplicates work. Also, due to primitive or user-defined operators with side effects, this alternative is not always correct. We therefore do not attempt to automatically rewrite programs to avoid buffering.

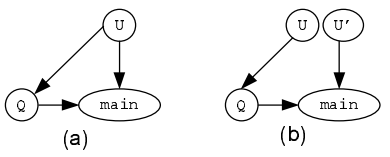

Figure 4: Buffering versus duplicating work.

Instead, DGL provides a manual construct to prevent buffering at a given node. If the name of an iterator variable is preceded by the '*' sign, the corresponding iterator is not shared by multiple consumers, but a new instance is generated for each consumer. The above example can thus be rewritten as:

```
LET  *U = Step(1, 10000, 1),
     Q = Query("complex query", U)
IN   U ++ Q
```

which is internally converted into the program shown earlier. When duplicating nodes with distribution primitives (e.g., Uniform), the compiler carefully seeds each primitive's random generator with the same value, to ensure that multiple executions return the same rows. For complex or user-defined iterators, it is responsibility of the programmer to ensure correctness when duplicating iterators.

2.5 Implementation Details

Figure 5 shows how a DGL program is compiled. Our compiler takes a DGL program and transforms it into intermediate C++ code. We then compile the intermediate C++ code and link it together with the DGL runtime library and any user-defined library (see Section 2.5.1). The resulting executable is then run to create and populate a database.

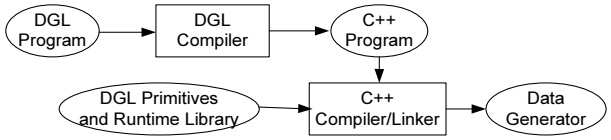

Figure 5: Building a data generator in DGL.

We could have implemented DGL on top of an existing functional language like Haskell (http://www.haskell.org). There were, however, some practical drawbacks with this approach. First, we wanted to explicitly fine-tune the internal mechanisms to evaluate expressions (e.g., iterator buffering) which are fixed in a language like Haskell. Also, there is currently no widely used interface between Haskell and RDBMSs (specifically for bulk-loading, which is crucial in our scenario). Finally, although DGL is functional in flavor, users usually define their own iterators using imperative languages like C++, and it

```
void aggSum::open()
local (I:Iterator)
01 I.open();
02 outputRow = new Row( dim(I.outputRow) );
03 for (i = 0; i < dim(outputRow); i++)
04   outputRow[i] = 0;

bool aggSum::getNext()
local (I:Iterator)
01 moreResults = I.getNext()
02 if (moreResults) outputRow += I.outputRow;
03 return moreResults;
```

Figure 6: Defining a new primitive iterator.

is easier to integrate components written in the same language.

2.5.1 Defining new Primitives

To add a new primitive iterator to DGL, we implement in C++ a derived sub-class from the base class `Iterator` that defines the methods `open` and `getNext`, and compile it into a new library, which is subsequently linked into the final executable. We next show how to implement a new iterator. Consider `aggSum`, which incrementally returns the sum of all prefixes of its input iterator. If `I=Step(1, 10, 1) ++ [2]`, then `aggSum(I)` returns `{[1,2], [3,4], [6,6], [10,8], ..., [45,18]}`. Pseudo-code for `aggSum` is shown in Figure 6. In our implementation, the base class `Iterator` defines a variable `outputRow` of type `Row` that holds the current row and `getNext` returns a boolean value indicating whether a new row was generated or an end-of-iterator was reached. The details are slightly more complex due to buffering, but we omit those for clarity.

2.5.2 Interface with the RDBMS

We use `ODBC` to connect to the RDBMS and execute queries specified by the `Query` iterator (`Query` is a thin wrapper on top of `ODBC`, which natively supports the iterator model). The `Persist` operator uses `ODBC`'s minimally logged bulk-loading extensions to maximize performance.

2.5.3 Iterator Buffering

As explained in Section 2.4, an iterator might have multiple consumers requesting rows at different rates. Since each consumer must obtain the same sequence of rows from the shared iterator, we must buffer all rows sent to fast consumers until the slowest consumer requests them. We keep an adaptive circular buffer of rows associated to each iterator (see Figure 7). The buffer maintains a window of the last rows produced by the iterator (e.g., rows R_3 to R_7 in the figure). Each consumer points to the last row it obtained from the buffer (e.g., consumer C_2 already read rows R_1 to R_5). When a consumer requests a new row and it does not point to the last valid row in the buffer, the

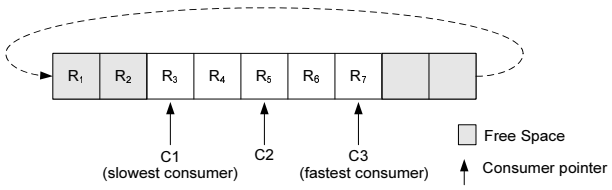

Figure 7: Sharing iterators among several consumers.

consumer's pointer is advanced and the corresponding row is returned (e.g., if C_2 requests a new row, its pointer moves to and returns row R_6). If instead the consumer points to the last valid row in the buffer (e.g., C_3 in the figure), a new row is first produced and stored in the circular buffer, and then returned to the consumer. When the slowest consumer requests a row (e.g., C_1 requesting row R_3), the row's slot in the buffer is freed and reused. Now suppose C_3 requests five new rows before C_1 requests any. In this case, there is no available buffer space for the fifth row. We then create a new buffer twice as big as the original one and move all the elements to this new buffer (ensuring constant amortized insertion time). When the slowest consumer requests a row and the buffer capacity falls below 25 percent, we replace the buffer by a new one half its original size (ensuring constant amortized deletion time). This adaptive behavior of the buffer is not enough in general, since one consumer can always be faster than another, and the size of the circular buffer would grow unbounded. In this case we swap the buffer to disk after it has grown beyond a certain size. We keep the original buffer in memory, but all subsequent "insertions" are written into a temporary sequential file. After consumers exhaust the in-memory portion of the buffer, they continue scanning the temporary file (we do not use the RDBMS capabilities for buffering because the OS file system is a light-weight and flexible alternative for this specific purpose). To avoid unnecessarily large temporary files, when the slowest consumer starts scanning the file, we create a new temporary file for all subsequent insertions. When the slowest consumer reads the last row of the current temporary file, the file is deleted. For many real examples, though, consumers request data synchronously, so there is virtually no impact due to buffering.

3 Annotated Schemas Using DGL

In this section we discuss a thin layer on top of DGL that allows *annotating* the `SQL CREATE TABLE` statement to additionally specify how to populate the created table. The syntax is as follows:

```
CREATE TABLE T (col_1 type_1, ..., col_n type_n)
[ other CREATE TABLE options ]
POPULATE N AS ( (col_1^1, ...  col_{n_1}^1) = expr_1,
                  ...
                (col_1^k, ...  col_{n_k}^k) = expr_k  )
```

where N is an integer that specifies an upper bound on the size of the created table, each column col_j in T is mentioned exactly once in the POPULATE clause, and $expr_i$ is a DGL expression with some additional syntactic sugar (see below). A database specification consists of a batch of annotated CREATE TABLE statements and it is processed as follows. First, each table is created omitting the POPULATE clause in its CREATE TABLE statement. Then, a single DGL program is built from all the POPULATE annotations. Finally, the DGL program is evaluated, populating database tables as a side effect.

In addition to plain DGL expressions, each $expr_i$ in a CREATE TABLE statement can refer to columns of any other table in the batch (including the table that $expr_i$ is populating) as if they were iterators. Temporary columns that are not part of the created table can be specified as well. Finally, Query iterators can refer to any table in the batch and also to the additional column id that is generated automatically by Persist. The only restriction is that the DAG associated to the resulting DGL program must be acyclic.

We now discuss through a series of examples how a DGL program is automatically generated from a batch of CREATE TABLE statements, significantly simplifying the task of the end-user. Consider the following simple specification:

```
CREATE TABLE Test (a INT, b INT, c INT, d INT)
POPULATE 10000 AS (
    (a, d) = myFunc(100),
    b = c - 1,
    c = a + d  )
```

The generated DGL program defines one iterator for each expression in the specification above, combines each iterator in the right column order, truncates the result to 10000 rows, and persists it into table Test. Columns used within expressions are referred to as projections of the corresponding iterators. The resulting program is as follows:

```
LET  Test_ad = myFunc(100),
     Test_c = Test_ad[0] + Test_ad[1],
     Test_b = Test_c - 1,
     Test = Top( Test_ad[0] ++ Test_b ++
                 Test_c ++ Test_ad[1],
                10000)
IN Persist(Test, "Test")
```

Evaluating this DGL program persists the specified data distribution into table Test. A set of multiple CREATE TABLE statements is treated in the same way. A single program is generated and the main expression combines each individual Persist primitive using the operator ++. In this way DGL can handle specifications in which some columns depend globally on the set of values of other columns, as shown in the next example (when Query primitives refer to the tables being populated, an additional mapping is applied).

```
CREATE TABLE R (a INT, b INT, c INT, d INT)
POPULATE 10000 AS (
  (a,b) = myFunc1(10),
  c = myFunc2(20),
  d = myFunc3(30)  )

CREATE TABLE S (f INT, g INT)
POPULATE 5000 AS (
  f = myFunc4(40),
  g = Query("SELECT AVG(b+c)
             FROM R
             GROUP BY a") )
```

In this situation we generate an additional expression that combines columns R.a, R.b, and R.c (we place a Top operator to ensure that the right number of rows is persisted) and persist it as a temporary table in the database. This temporary table is then used by the query that generates column S.g[4]. Note that we only temporarily persist R.a, R.b, and R.c instead of using the final populated table R to allow specifications in which columns from two tables mutually depend on each other without forming a cycle in the DAG. Of course, if there is a partial order for the creation of tables in the batch, a simple optimization is possible in which the intermediate table is not created at all.

Additionally, we use an optimization that avoids buffering the iterators for columns R.a, R.b, and R.c, since they would be already persisted in the RDBMS due to S.g. For that purpose, we create a proxy iterator that simply performs a sequential scan over the temporary table created by Persist. Each original consumer of columns R.a, R.b, or R.c is changed so that it consumes rows from this proxy iterator. The resulting program for the above specification is shown next.

```
LET R_ab = myFunc1(10),
    R_c = myFunc2(20),
    R_d = myFunc3(30),
    S_f = myFunc4(40),

    tmp1 = Persist(Top(R_ab ++ R_c ), 10000),
    tmp1Proxy = Query ( "SELECT * FROM <<0>>",
                        tmp1 ),
    S_g = Query ( "SELECT AVG(v1+v2)
                   FROM <<0>>
                   GROUP BY <<0>>.v0",
                  tmp1 )
    R = Top( tmp1Proxy ++ R_d, 10000),
    S = Top( S_f ++ S_g, 5000),
IN Persist(R, "R") ++ Persist(S, "S")
```

Note that the initial annotation on the CREATE TABLE statement is much simpler to understand and write than the corresponding DGL program. In Section 4 we show several examples that use annotated schemas to specify complex database distributions.

[4]If the input to a Query operator is already "Persisted", we do not create a second copy in the database, but reuse the original one.

4 Evaluation

We now illustrate how to generate several data distributions previously proposed in the literature using DGL annotations.

4.1 Benchmarks

We start by showing how to populate the relations of three benchmarks described in [11]. We note that the relations of these benchmarks are somewhat simple to generate and mostly produce tables with no complex correlations.

Wisconsin

The Wisconsin Benchmark [5] was proposed in the early eighties to test the performance of the major components of a RDBMS, using relations with well-understood semantics and statistics. The following is a partial specification using DGL to generate table TENKTUP of this benchmark.

```
CREATE TABLE TENKTUP ( ... )
POPULATE 10000 AS (
  unique1 = Permutation(10000),
  unique2 = Step(0, 10000, 1),
  four = unique1 mod 4,
  onePercent = unique1 mod 100,
  oddOnePercent = onePercent * 2 + 1, ... )
```

where Permutation is defined as follows:

```
Permutation(n) =
  LET tmp= PERSIST(Uniform(0, 1))
  IN Query("SELECT id FROM <0> ORDER BY v0",tmp)
```

In [11, 12] an alternative procedure is used to generate permutations, which is based on congruential generators that return dense uniform distributions. While this alternative is less costly (and we could have easily implemented it in DGL) it can only be used to generate *one* different permutation (others are just circular shifts). In our experiments we use the slower but more flexible alternative described above.

AS^{3}AP

The AS^{3}AP Benchmark [17] is a successor of the Wisconsin Benchmark and gives a more balanced and realistic evaluation of the performance of a RDBMS. In addition to the tests performed by the Wisconsin Benchmark, AS^{3}AP tests utility functions, mix batch and interactive queries and emphasize multiuser tests. From a data generation perspective, AS^{3}AP introduces non-uniform distributions in some columns, but columns remain independently generated from each other. We now show a partial DGL specification for the UPDATES table in the AS^{3}AP benchmark.

```
CREATE TABLE UPDATES ( ... )
POPULATE 10000 AS (
  key = 1 + Permutation(10000),
  signed= 100000 * Permutation(10000) - 50000000,
  float= 100000 * Zipfian(1.0, 10) - 500000000,
  double= Normal(1, 0), 0, 1, [0], ... )
```

We note that the actual specification states that the value 1 must not appear in column key to allow "not-found-scans" for a value within the range of the attribute. For simplicity, we omit the value 0 instead, which conveys the same functionality but makes the specification slightly simpler. Both columns signed and float are specified to be sparse, so that the distinct values are stretched to the range $[-5 \cdot 10^8, 5 \cdot 10^8]$ and thus can be used to phrase queries with relative selectivities that are a function of the database size.

Set Query

The Set Query Benchmark [13] was designed to measure the performance of a new class of systems that exploit the strategic value of operational data in commercial enterprises. While the queries in the Set Query benchmark are complex, the data generation program is surprisingly simple (specifically, each column is populated independently with uniformly distributed integer values). We now show the DGL specification for the BENCH table in this benchmark.

```
CREATE TABLE BENCH ( ... )
POPULATE 1000000 AS (
   KSeq = Step(1, 1000000, 1),
   K500K = UniformInt(1, 500000),
   K250K = UniformInt(1, 250000),
   ...,
   K2 = UniformInt(1, 2)
)
```

4.2 Research papers

In this section we present an –incomplete– sample of data distributions recently used in the literature to validate novel cardinality estimation techniques.

The M-Gaussian synthetic distribution [7, 10, 15] consists of a predetermined number of overlapping multidimensional gaussian bells. The parameters for this distribution are: the domain for the gaussian centers (Lo, Hi), the number of gaussian bells p, the standard deviation of each gaussian distribution sigma, and a Zipfian parameter z that regulates the total number of rows contained in each gaussian bell. This distribution is specified below:

```
M-Gaussian(Lo, Hi, sigma, z, p) =
  LET centerList = Top( Uniform(Lo, Hi), p )
      indexes = Zipfian(z, p),
      centers = TableApply(centerList, indexes)
  IN Normal(centers, sigma)
```

where centerList generates p random gaussian centers, indexes generates N indexes (which point to some center) and centers is an iterator that returns a stream of centers taken from centerList (for efficiency we store

the gaussian centers in an in-memory table, since by definition there are a small number of them). Finally, we apply a `Normal` transformation to the centers to obtain the desired distribution. Using the function defined above, the two dimensional data set used in [7] is generated as:

```
CREATE TABLE Test ( x REAL, y REAL, z REAL )
POPULATE 1000000 AS (
  (x,y,z)= M-Gaussian([0,0,0], [1000,1000,1000],
                      [25,25,25], 1.0, 25 )
)
```

M-Zipfian distributions were introduced in [14] and subsequently used in [1, 7], among others. Each dimension has a number of distinct values, and the value sets of each dimension are generated independently. Frequencies are generated according to a Zipfian distribution and assigned to randomly chosen cells in the joint frequency distribution matrix. The following DGL function returns cell indexes for a two-dimensional Multi-Zipfian distribution:

```
M-Zipfian2D ( N1, N2, z ) =
  LET indexes = Zipfian(z, N1*N2),
      mapIndexes= TableApply(Permutation(N1*N2),
                             indexes)
  IN mapIndexes/N2 ++ mapIndexes%N2
```

where `indexes` chooses a random number between 1 and the number of cells in the joint distribution following a Zipfian distribution, `mapIndexes` applies a random permutation to the values in `indexes`, and the main expression unfolds each number in the two-dimensional coordinates of a cell. The result of `M-Zipfian2D` is a stream of cell-indexes, which can then be mapped to a valid element in the data domain.

A different data generation procedure derived from Zipfian distributions was used in [9]. Essentially, the data generation program [8] produces `N` rows in `d` consecutive clusters of values whose frequencies follow a Zipfian distribution. A simple `DGL` specification for this generator is:

```
TPCH-Zipf ( N, z, d ) =
  Duplicate( Step(1, d, 1), ZipfianD(N, z, d) )
```

where `ZipfianD` generates at each step the *i-th* frequency of an ideal Zipfian distribution with parameter `z` ($1 \leq i \leq d$) for a total of `N` rows.

In the context of statistics on query expressions [6], we needed to generate data that exhibited dependencies between filter and join predicates. We used a generator that populated a fact table `R` with a foreign-key to a dimension table `S`. We wanted that the number of matches in `S` from the foreign-key join from `R` follow a Zipfian distribution, and also that a column `c` in `S` maintained the number of elements in `R` that were joined with the corresponding row in `S`. The following `DGL` specification generates such distribution:

```
CREATE TABLE R (r INT, s REAL, ...)
POPULATE 1000000 AS (
  r = Step(1, 1000000,1),
  s = Zipfian(1.0, 50000),
  ... )

CREATE TABLE S (s INT, c INT, ...)
POPULATE 50000 as (
  (s, c) = Query("SELECT s, count(*)
                  FROM R
                  GROUP BY s"),
  ... )
```

where the `Query` iterator in `S` returns all distinct values in `R` as well as their counts, which is precisely what we wanted to generate.

4.3 Complex Dependencies

TPC-H [16] is a decision support benchmark consisting of business-oriented data and ad-hoc queries. Data populating the database has been chosen to have broad industry-wide relevance while maintaining a sufficient degree of ease of implementation. While TPC-H defines rich schema, the standard data generation tool is rather simple. Almost all columns are uniformly generated and, and with some exceptions, all columns are uncorrelated. (The exceptions are `o_totalprice`, which is functionally determined by `l_discount`, `l_extendedprice`, and `l_tax`, and `date` columns in table `lineitem`, which satisfy precedence constraints.) We next use the schema of TPC-H and show how to specify some complex dependencies for a TPC-H like database. For clarity, we present small fragments of `DGL` instead of the full specification.

Figure 8(a) specifies that order arrivals follow a Poisson distribution starting in `'1992/01/01'`. We use a `Poisson` iterator that returns inter-arrival times and aggregate it using `aggSum` (defined in Section 2.5.1). Finally, we add `'1992/01/01'` to the resulting iterator.

Figure 8(b) models the fact that the number of line items for a given order follows a Zipfian distribution (i.e., there are some very large orders and many small ones). Additionally, the ship date of an item occurs after $1 \leq k \leq 10$ days of the order date, where k follows a Zipfian distribution with $z = 1.75$. Finally, the commit and receipt dates of an item follow a two-dimensional normal distribution centered around 5 days after the ship date[5]. We define `l_orderkey` by selecting all keys from `orders` and duplicating them a certain number of times (specified by a Zipfian distribution). We specify a temporary column `tmpDate` that is not persisted into the database. Instead, we use `tmpDate` to define `l_shipdate`, which in turn defines both `l_commitdate` and `l_receiptdate`.

[5]Receipt and commit dates could be earlier than the ship date. To avoid this rare situations we can add a `MAX` operator so that `l_commitdate` and `l_receiptdate` always occur after `l_shipdate`.

Figure 8(c) shows how to model that the discount of each line item is correlated to the number of such parts sold globally. Specifically, let $|P|$ be the number of parts equal to that of lineitem that are globally sold. If $|P|$ is beyond 1000, the discount for the lineitem is 25%. Otherwise, the discount is $|P| * 0.025\%$. We use a `Query` iterator that computes the total number of each distinct part value in `LINEITEM`, and then join this "aggregated" table with the partially generated `LINEITEM`, computing the discount of each row. Note the final order clause in the `Query` iterator (`ORDER BY L.id`). This is required to guarantee that the "discount" iterator is in sync with the other columns in lineitem, since the join in the `Query` iterator might be non order-preserving.

Assuming that customers pay an order whenever it is closed, we define the debt of a customer as the total price of all their still-open orders. Figure 8(d) specifies that the 100 customers with the largest debt have a balance that is normally distributed around three times their respective debts with a standard deviation of 25000. The remaining customers' balances follow a normal distribution around half its debt with a standard deviation of 500. We first generate the customer keys using a `Query` iterator that additionally returns the "debt" of each customer in a temporary column `tmpDebt`. We then generate `c_acctbal` as the union of two iterators. The first one gets the top 100 rows from `tmpDebt` and produces the corresponding normal distribution, while the second one does the same to `tmpDebt`'s 101-*th* row and beyond (using iterator `Skip`).

Finally, Figure 8(e) models the fact that all parts in an order are sold by suppliers that live in the same nation as the customer. For this example we assume that `l_orderkey` was already generated with some distribution and we generate the complementary distributions for `l_suppkey` (a random supplier from the same nation as the orders' customer), and `l_partkey` (a random part from that supplier). We first define a temporary column `tmpNation` which consists of the nations of the corresponding orders' customers. Then, we define `l_suppkey` with a `Query` iterator that uses the extended `SQL CROSS APPLY` and `newId` operators (`CROSS APPLY` invokes a table-valued function for each row in the outer table expression and returns a unified result set out of all of the partial table-valued results, while `newId` returns a random identifier for each row in the result). This iterator selects at random one row from `SUPPLIER` that has the same nation as each row in `tmpNation`. A similar iterator selects a random part from each element in `l_suppkey`.

```
CREATE TABLE ORDERS (...) POPULATE N AS (
   o_orderdate = '1992/01/01' + aggSum( Poisson(5) ),
   ... )
```
(a)

```
CREATE TABLE LINEITEM (...) POPULATE N AS (
   (l_orderkey, tmpDate) = Duplicate(
        Query("SELECT o_orderkey, o_orderdate FROM ORDERS"),
        Zipfian(1.0, 1000) ),
   l_shipdate = tmpDate + Zipfian(1.75, 10),
   (l_commitdate, l_receiptdate) = Normal([0,0],[1,1]) +
                          [5 + l_shipdate, 5 + l_shipdate],
   ... )
```
(b)

```
CREATE TABLE LINEITEM (...) POPULATE N AS (
    l_discount =
        Query("SELECT CASE WHEN pTotals.pCount>1000 THEN 0.25
                           ELSE pTotals.pCount*0.00025
                      END
               FROM LINEITEM L,
                    ( SELECT l_partkey, COUNT(*) as pCount
                      FROM LINEITEM
                      GROUP BY l_partkey ) as pTotals
                      WHERE L.l_partkey = pTotals.l_partkey
                      ORDER BY L.id"),
  ... )
```
(c)

```
CREATE TABLE CUSTOMER (...) POPULATE N AS (
  (c_custkey, tmpDebt) =
     Query("SELECT o_custkey, sum(o_totalprice) as sumPrice
            FROM ORDERS
            WHERE o_orderstatus='O'
            GROUP BY o_custkey
            ORDER BY sumPrice desc")
  c_acctbal = Union ( Normal(  Top(TmpDebt, 100) * 3, 25000 ),
                      Normal( Skip(TmpDebt, 100) / 2, 500 )
                    )
  ... )
```
(d)

```
CREATE TABLE LINEITEM (...) POPULATE N AS (
  tmpNation = Query(" SELECT c_nationkey
                      FROM LINEITEM, ORDERS, CUSTOMER
                      WHERE l_orderkey=o_orderkey
                        AND o_custkey=c_custkey
                      ORDER BY LINEITEM.id " ),
  l_suppkey = Query(" SELECT S.s_suppkey
                      FROM LINEITEM CROSS APPLY (
                        SELECT TOP 1 s_suppkey
                        FROM SUPPLIER
                        WHERE s_nationkey = tmpNation
                        ORDER BY newId() ) as S "),
  l_partkey = Query(" SELECT PS.ps_partkey
                      FROM LINEITEM CROSS APPLY (
                        SELECT TOP 1 ps_partkey
                        FROM PARTSUPP
                        WHERE ps_suppkey = s_suppkey
                        ORDER BY newId() ) as PS" )
  ... )
```
(e)

Figure 8: `DGL` specifications for complex correlations.

4.4 Data Generation

We next show preliminary results of actual data generation runs to analyze the performance of `DGL`. Figure 9 shows, for each of the examples in the previous section, the total time required to create and populate 1GB worth of data (the "*Complex*" database in the figure implements all the partial specifications in Figure 8). We used Microsoft SQL Server as the underlying RDBMS and measured the time it took to populate the different tables without building any index structures. We see that all generators but *Complex* finish in under 7 minutes (note that *Baseline* populates a dummy 1GB database with constant values, so its execution time is a lower bound to generate a 1GB database). The reason that *Complex* takes around 13 minutes to finish is that due to complex correlations, some large intermediate results must be materialized before generating the final tables (around 4 minutes were spent materializing temporary tables).

Database	Rows	Time	MB/sec
Baseline	-	3'37"	4.72
Wisconsin	4.5M	4'11"	4.08
AS^{3}AP	9.9M	6'42"	2.55
Set Query	15.6M	5'08"	3.32
M-Gaussian	42M	5'51"	2.92
M-Zipfian	50M	6'25"	2.66
JoinCorr	48M	6'02"	2.83
Complex	10.5M	13'12"	1.29

Figure 9: Generating 1GB synthetic databases.

5 Conclusion

In this paper we introduced `DGL`, a simple specification language to generate databases with complex synthetic distributions and inter-table correlations. We showed that many synthetic distributions proposed earlier in the literature can be easily specified using `DGL`. We also showed that the resulting data generators are efficient. We believe that `DGL` is an important first step towards reusable synthetic databases.

References

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1999.

[2] A. Aboulnaga, J. Naughton, and C. Zhang. Generating synthetic complex-structured XML data. In *In Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2001.

[3] S. Agrawal, S. Chaudhuri, and V. Narasayya. Materialized view and index selection tool for Microsoft SQL Server 2000. In *Proceedings of the ACM International Conference on Management of Data*, 2001.

[4] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: a template-based data generator for XML. In *In Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2002.

[5] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking database systems: A systematic approach. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1983.

[6] N. Bruno and S. Chaudhuri. Conditional selectivity for statistics on query expressions. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[7] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A multidimensional workload-aware histogram. In *Proceedings of the ACM International Conference on Management of Data*, 2001.

[8] S. Chaudhuri and V. Narasayya. TPC-D data generation with skew. Accesible via `ftp` at `ftp.research.microsoft.com/users/viveknar/tpcdskew`.

[9] S. Chaudhuri and V. Narasayya. Automating statistics management for query optimizers. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2000.

[10] D. Donjerkovic, Y. E. Ioannidis, and R. Ramakrishnan. Dynamic histograms: Capturing evolving data sets. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2000.

[11] J. Gray. *The Benchmark Handbook for Database and Transaction Systems.* Morgan Kaufmann, 1993.

[12] J. Gray et al. Quickly generating billion-record synthetic databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1994.

[13] P. E. O'Neil. A set query benchmark for large databases. In *Proceedings of the 15th International Computer Measurement Group Conference*, 1989.

[14] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1997.

[15] N. Thaper et al. Dynamic multidimensional histograms. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2002.

[16] TPC Benchmark H. Decision support. *Available at* `http://www.tpc.org/tpch/spec/tpch2.1.0.pdf`.

[17] C. Turbyfill, C. Orju, and D. Bitton. ASAP: A comparative relational database benchmark. In *Proceedings of Compcon*, 1989.

[18] G. Valentin et al. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2000.