Touchstone: Generating Enormous Query-Aware Test Databases

Yuming Li[1], Rong Zhang[1]*, Xiaoyan Yang[2], Zhenjie Zhang[2], Aoying Zhou[1]
[1]*East China Normal University, liyuming@stu.ecnu.edu.cn, {rzhang, ayzhou}@sei.ecnu.edu.cn*
[2]*Singapore R&D, Yitu Technology Ltd., {xiaoyan.yang, zhenjie.zhang}@yitu-inc.com*

Abstract

Query-aware synthetic data generation is an essential and highly challenging task, important for database management system (DBMS) testing, database application testing and application-driven benchmarking. Prior studies on query-aware data generation suffer common problems of limited parallelization, poor scalability, and excessive memory consumption, making these systems unsatisfactory to terabyte scale data generation. In order to fill the gap between the existing data generation techniques and the emerging demands of enormous query-aware test databases, we design and implement our new data generator, called *Touchstone*. *Touchstone* adopts the random sampling algorithm instantiating the query parameters and the new data generation schema generating the test database, to achieve fully parallel data generation, linear scalability and austere memory consumption. Our experimental results show that *Touchstone* consistently outperforms the state-of-the-art solution on TPC-H workload by a 1000× speedup without sacrificing accuracy.

1 Introduction

Query-aware test databases are essential to database testings for analytical applications, helping the database developers and users to better evaluate and understand the performance of the system in various cases. The applications of query-aware data generators include DBMS testing, database application testing and application-driven benchmarking [5, 15]. For example, during the database selection and performance optimization, the internal databases in production are hard to be shared for performance testing due to the privacy considerations, so we need to generate synthetic databases with the similar workload characteristics of the target queries. A bulk of existing data generators, e.g., [12, 11, 4, 21], generate test databases independent of the test queries, which only consider the data distribution of inter- and intra-attribute. They fail to guarantee the similar workload characteristics of the test queries, therefore it's difficult to match the overheads of the query execution engine for real world workloads. A number of other studies, e.g., [6, 14, 5, 15], attempt to build query-aware data generators. But the performance of the state-of-the-art solution MyBenchmark [15] still remains far from satisfactory, due to the lack of parallelization, scalability and memory usage control, as well as the narrow support of non-equi-join workload. In order to generate the enormous query-aware test databases, we design and implement *Touchstone*, a new query-aware data generator, based on the following design principles:

Full Parallelization: With the explosive growth of data volume in the industrial applications, the database system is expected to support storage and access services for terabyte or even petabyte scale data. So the synthetic data generator must be fully parallel for generating such extremely large test databases.

Linear Scalability: The single machine has been far from meeting the requirements of generating large test databases, and the data scales may be unbelievably big for the future applications, therefore the data generator needs to be well scalable to multiple nodes and data size.

Austere Memory Consumption: When generating the synthetic database for multiple queries, memory could easily be the bottleneck, because massive information is maintained by the data generator in order to guarantee the dependencies among columns. The memory usage needs to be carefully controlled and minimized.

Since [6, 14, 5, 15] are closest to the target of this work, we list the following key insufficiencies of these studies for elaborating the necessary of proposing *Touchstone*. In particular, all of these approaches do not support fully parallel data generation in a distributed environment due to the primitive data generation algorithms, limiting the efficiency of data generation over target size at terabytes. Moreover, their memory consumptions, e.g., symbolic databases of QAGen [6], constrained bipartite graphs of WAGen [14] and MyBenchmark [15], inverted indexes for generating foreign keys of DCGen [5], strongly depend on the size of generation outputs. Once the memory is insufficient to host the complete intermediate state, huge computational resources are wasted on disk I/O operations. One key advantage of our work is the support of non-equi-join workload, which is important for real world applications but not supported by any of the existing approaches. Last but not least, different from the approaches in [6, 14, 15], which can not

*Rong Zhang is the corresponding author.

guarantee to generate one database instance (DBI) for multiple queries, our approach is capable of generating one single DBI for a group of test queries.

In query-aware data generation, we need to handle the extremely complicated dependencies among columns which are produced by the complex workload characteristics specified on the target test queries, as well as the data characteristics specified on the columns. *Touchstone* achieves fully parallel data generation, linear scalability and austere memory consumption for supporting the generation of enormous query-aware test databases. There are two core techniques employed by *Touchstone* beneath the accomplishments of all above enticing features. Firstly, *Touchstone* employs a completely new query instantiation scheme adopting the random sampling algorithm, which supports a large and useful class of queries, as well as the actual query execution trees from the running database system. Secondly, *Touchstone* is equipped with a new data generation scheme using the constraint chain data structure, which easily enables thread-level parallelization. This scheme allows *Touchstone* to easily scale up to dozes of nodes and hundreds of cores for parallelized database generation. In summary, *Touchstone* is a scalable query-aware data generator with a wide support to complex queries of analytical applications, and achieves a 1000× performance improvement against the state-of-the-art work MyBenchmark [15].

The rest of the paper is organized as follows: Section 2 provides the preliminaries including problem definition of the query-aware data generation, overview of the system architecture and introduction to the random column generator of *Touchstone*. Section 3 introduces the techniques of query instantiation in *Touchstone*. Section 4 covers the scheme of distributed data generation used by *Touchstone*. Section 5 reports the experimental results of *Touchstone* over benchmark workloads. Section 6 reviews the related studies in the literature. Section 7 finally discusses the limitations and privacy issues of *Touchstone*, and concludes the paper.

2 Preliminaries

In this section, we present the problem definition of query-aware data generation, formalize the input and output format of *Touchstone*, describe the overall architecture of *Touchstone*, and finally introduce the random column generator used in *Touchstone*.

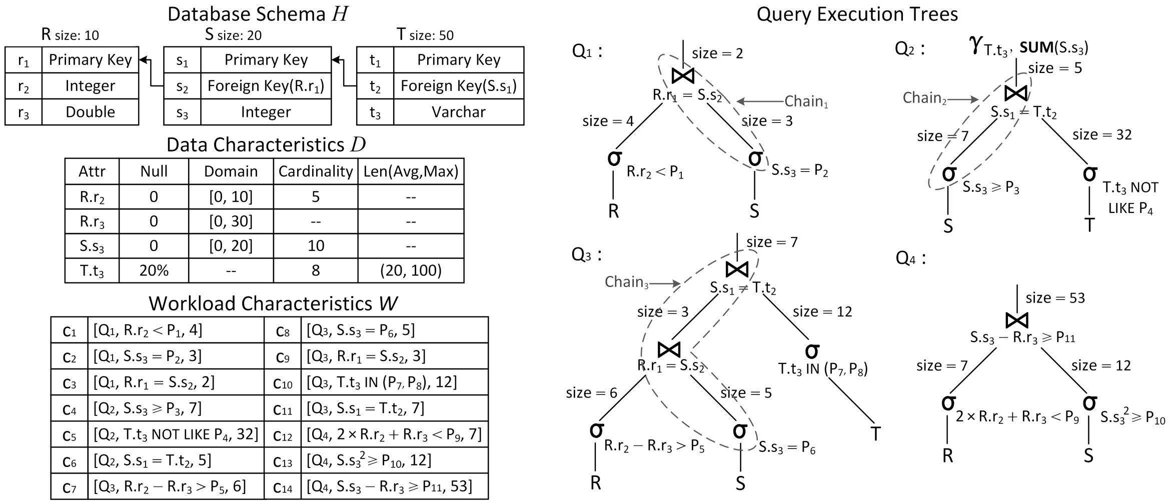
2.1 Problem Formulation

The input of *Touchstone* includes database schema H, data characteristics D and workload characteristics W, as illustrated in Figure 1. H defines data types of columns, primary key and foreign key constraints. In Figure 1, there are three tables R, S, and T. For example, table S has 20 tuples and three columns, including a primary key $S.s_1$, a foreign key $S.s_2$ referenced to $R.r_1$ from table R, and an integer typed column $S.s_3$. Data characteristics D of columns are defined in a meta table, in which the user defines the percentage of *Null* values, the domain of the column, the cardinality of unique values and the average length and maximum length for varchar typed columns. In our example, the user expects to see 5 unique values on column $R.r_2$ in the domain [0, 10], and 8 different strings with an average length of 20 and a maximum length of 100 for column $T.t_3$ with 20% *Null* values. Workload characteristics W are represented by a set of parameterized queries which are annotated with several cardinality constraints. In Figure 1, our sample input consists of four parameterized queries, i.e., $Q = \{Q_1, Q_2, Q_3, Q_4\}$. These four queries contain 11 variable parameters, i.e., $P = \{P_1, P_2, ..., P_{11}\}$. Each filter/join operator in the queries is associated with a size constraint, defining the expected cardinality of the processing outcomes. Therefore, there are 14 filter/join operators and corresponding 14 cardinality constraints in our example, i.e., $C = \{c_1, c_2, ..., c_{14}\}$. If we zoom into query Q_1, there are three cardinality constraints, two of which have variable parameters, i.e., $R.r_2 < P_1$ and $S.s_3 = P_2$ with parameters P_1 and P_2 respectively. Our target is to generate the three tables and instantiate all the variable query parameters. In the following, we formulate the definition of cardinality constraints.

Definition 1 ***Cardinality Constraint:*** *Given a filter (σ) or join ($\bowtie$) operator, a cardinality constraint c is denoted as a triplet c = [Q, p, s], where Q indicates the involving query, p gives the predicate on the incoming tuples, and s is the expected cardinality of operator outcomes.*

The cardinality constraint c_1 in Figure 1, for example, is written as $[Q_1, R.r_2 < P_1, 4]$, indicating that the operator with predicate $R.r_2 < P_1$ in query Q_1 is expected to output 4 tuples. For conjunctive and disjunctive operators, their cardinality constraints can be split to multiple cardinality constraints for each basic predicate using standard probability theory. These cardinality constraints generally characterize the computational workload of query processing engines, because the computational overhead mainly depends on the size of the data in processing. This hypothesis is verified in our experimental evaluations.

While the focus of cardinality constraints is on filter and join operators, *Touchstone* also supports complex queries with other operators, including aggregation, groupby and orderby. For example, the query Q_2 in Figure 1 applies groupby operator on $T.t_3$ and summation operator on $S.s_3$ over the grouped tuples. The cardinality of the output tuples from these operators, however, is



Database Schema H

R size: 10

r_1	Primary Key
r_2	Integer
r_3	Double

S size: 20

s_1	Primary Key
s_2	Foreign Key(R.r_1)
s_3	Integer

T size: 50

t_1	Primary Key
t_2	Foreign Key(S.s_1)
t_3	Varchar

Data Characteristics D

Attr	Null	Domain	Cardinality	Len(Avg,Max)
R.r_2	0	[0, 10]	5	--
R.r_3	0	[0, 30]	--	--
S.s_3	0	[0, 20]	10	--
T.t_3	20%	--	8	(20, 100)

Workload Characteristics W

c_1	[Q_1, R.$r_2 < P_1$, 4]	c_8	[Q_3, S.$s_3 = P_6$, 5]
c_2	[Q_1, S.$s_3 = P_2$, 3]	c_9	[Q_3, R.r_1 = S.s_2, 3]
c_3	[Q_1, R.r_1 = S.s_2, 2]	c_{10}	[Q_3, T.t_3 IN (P_7, P_8), 12]
c_4	[Q_2, S.$s_3 \geqslant P_3$, 7]	c_{11}	[Q_3, S.s_1 = T.t_2, 7]
c_5	[Q_2, T.t_3 NOT LIKE P_4, 32]	c_{12}	[Q_4, $2 \times$ R.r_2 + R.$r_3 < P_9$, 7]
c_6	[Q_2, S.s_1 = T.t_2, 5]	c_{13}	[Q_4, S.$s_3^2 \geqslant P_{10}$, 12]
c_7	[Q_3, R.r_2 − R.$r_3 > P_5$, 6]	c_{14}	[Q_4, S.s_3 − R.$r_3 \geqslant P_{11}$, 53]

Figure 1: Example inputs of database schema, data characteristics and workload characteristics to *Touchstone*

mostly determined, if it does not contain a having clause. And such operators are usually engaged on the top of query execution tree, hence the output result cardinalities generally do not affect the total computational cost of query processing. Based on these observations, it is unnecessary to pose explicit cardinality constraints over these operators [14, 5] in *Touchstone*.

Based on the target operators (filter or join) and the predicates with equality or non-equality conditional expressions, we divide the cardinality constraints into four types, i.e., $C = C^{\sigma}_{=} \cup C^{\sigma}_{\neq} \cup C^{\bowtie}_{=} \cup C^{\bowtie}_{\neq}$. Accordingly, we classify the example constraints in Figure 1 as $C^{\sigma}_{=} = \{c_2, c_5, c_8, c_{10}\}$[1], $C^{\sigma}_{\neq} = \{c_1, c_4, c_7, c_{12}, c_{13}\}$, $C^{\bowtie}_{=} = \{c_3, c_6, c_9, c_{11}\}$ and $C^{\bowtie}_{\neq} = \{c_{14}\}$. Following the common practice in [5, 25, 26], the equi-join operator is *always* applied on the pair of primary and foreign keys.

Then we formulate the problem of query-aware data generation as follows.

Definition 2 ***Query-Aware Data Generation Problem:*** *Given the input database schema H, data characteristic D and workload characteristics W, the objective of data generation is to generate a database instance (DBI) and instantiated queries, such that 1) the data in the tables strictly follows the specified data characteristics D; 2) the variable parameters in the queries are appropriately instantiated; and 3) the executions of the instantiated queries on the generated DBI produce exactly the expected output cardinality specified by workload characteristics W on each operator.*

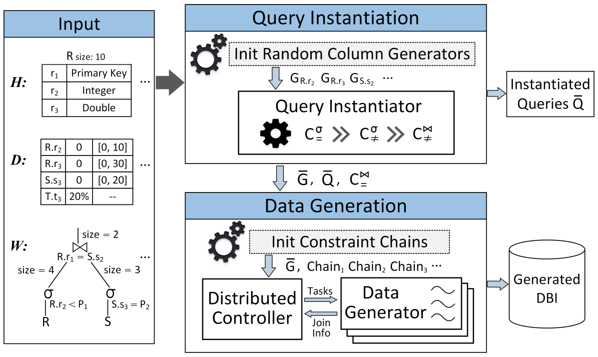
While the general solution to query-aware data generation problem (even verification on the validity of the constraints) is NP-hard [22], we aim to design a data generator, by relaxing the third target in the definition above. Specifically, the output DBI is expected to perform as closely as the cardinality constraints in C. Given the actual/expected cardinalities of processing outputs, i.e., $\{\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_n\}$, corresponding to constraints on the queries in $C = \{c_1, c_2, \ldots, c_n\}$, we aim to minimize the global relative error $\frac{\sum_{c_i \in C} |c_i.s - \hat{s}_i|}{\sum_{c_i \in C} c_i.s}$. Even if the user specified workload in W contains conflicted constraints, *Touchstone* still attempts to generate a DBI with the best effort. For any existing application, the inputs of data characteristics and workload characteristics can be automatically generated over the running database, and the relevant techniques are only concrete implementation works.

2.2 Overview of Touchstone

Based on the definition of query-aware data generation problem, the input to *Touchstone* contains database schema H, data characteristics D of non-key columns and workload characteristics W of test queries. The output of *Touchstone* consists of two parts, namely the instantiated queries and the generated DBI. To support the generation procedure, the infrastructure of *Touchstone* is divided into two major components, which are responsible for query instantiation and data generation respectively, as shown in Figure 2.

Query Instantiation: Given the inputs including database schema H, data characteristics D, *Touchstone* builds a group of random column generators for non-key columns, denoted by G, each G_i in which corresponds to a column of the target tables. Given the input workload

[1] If the relational operator in a selection predicate belongs to {=, !=, *in*, *not in*, *like*, *not like}*, then the corresponding cardinality constraint is classified as $C^{\sigma}_{=}$.


Input
R size: 10
H:
D:
W:
size = 2
R.r$_1$ = S.s$_2$
size = 4
size = 3
σ
σ
R.r$_2$ < P$_1$
S.s$_3$ = P$_2$
R
S
Query Instantiation
Init Random Column Generators
G$_{R.r_2}$ G$_{R.r_3}$ G$_{S.s_2}$ ···
Query Instantiator
$C^{\sigma}_{=}$ ≫ $C^{\sigma}_{\neq}$ ≫ $C^{\bowtie}_{\neq}$
Instantiated Queries $\bar{Q}$
$\bar{G}$, $\bar{Q}$, $C^{\bowtie}_{=}$
Data Generation
Init Constraint Chains
$\bar{G}$, Chain$_1$ Chain$_2$ Chain$_3$ ···
Distributed Controller
Tasks
Join Info
Data Generator
Generated DBI

r$_1$	Primary Key
r$_2$	Integer
r$_3$	Double

R.r$_2$	0	[0, 10]
R.r$_3$	0	[0, 30]
S.s$_3$	0	[0, 20]
T.t$_3$	20%	--

Figure 2: The overall architecture of *Touchstone*

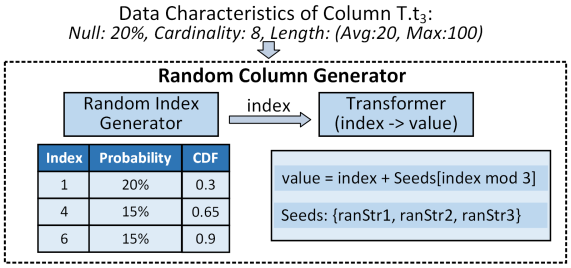
characteristics W, *Touchstone* instantiates the parameterized queries by adjusting the related column generators if necessary and choosing appropriate values for the variable parameters in the predicates of $c \in C^{\sigma}_{=} \cup C^{\sigma}_{\neq} \cup C^{\bowtie}_{\neq}$. The instantiated queries $\bar{Q}$ are output to the users for reference, while the queries $\bar{Q}$ and the adjusted column generators $\bar{G}$ are fed into the data generation component. The technical details are available in Section 3.

Data Generation: Given the inputs including instantiated queries $\bar{Q}$ and constraints over the equi-join operators $C^{\bowtie}_{=}$ specified in W, *Touchstone* decomposes the query trees annotated with constraints into constraint chains, in order to decouple the dependencies among columns, especially for primary-foreign-key pairs. Data generation component generally deploys the data generators over a distributed platform. The random column generators and constraint chains are distributed to all data generators for independent and parallel tuple generation. The technical details are available in Section 4.

2.3 Random Column Generator

The basic elements of *Touchstone* system are a group of random column generators $G = \{G_1, G_2, \dots, G_n\}$, which determine the data distributions of all non-key columns to be generated. A random column generator G_i in G is capable of generating values for the specified column, while meeting the required data characteristics *in expectation*. In the following, we give the detailed description of the random column generator, which is the foundation of algorithm designs introduced in the rest of the paper.

A random column generator G_i contains two parts, a random index generator and a value transformer as shown in Figure 3. In the random index generator, the size of output index domain is identical to the expected cardinality of unique values of the corresponding column, i.e., integer from 0 to $n-1$ while n is the specified cardinality. Given an index, the transformer deterministically maps it to a value in the specified domain of the column. We adopt different transformers based on the type of the column. For numeric types, e.g., *Integer*, we simply pick up a linear function which uniformly maps the index to the value domain. For string types, e.g., *Varchar*, there are some seed strings pre-generated randomly, which satisfy the specified length requirements. We first select a seed string based on the input index as shown in Figure 3, and then concatenate the index and the selected seed string as the output value. This approach allows us to easily control the cardinality of string typed columns with tiny memory consumption.


Data Characteristics of Column T.t$_3$:
Null: 20%, Cardinality: 8, Length: (Avg:20, Max:100)
Random Column Generator
Random Index Generator
index
Transformer (index -> value)

Index	Probability	CDF
1	20%	0.3
4	15%	0.65
6	15%	0.9

value = index + Seeds[index mod 3]

Seeds: {ranStr1, ranStr2, ranStr3}

Figure 3: An example generator for column $T.t_3$

To manipulate the distribution of the column values, there is a probability table in the random index generator. The probability table consists of a number of entries and each entry corresponds to an index. Specifically, each entry in the table (k_i, p_i, c_i) specifies an index k_i, the probability p_i of occurrence, and the cumulative probability c_i for all indexes no larger than k_i. In order to save the memory space, we compress the table by keeping only the entries with non-uniform probabilities. If an index does not appear in the probability table, its probability is automatically set by the uniform probability. The entries in the table are ordered by k_i. In Figure 3, we present an example of random column generator designed for column $T.t_3$ from example inputs in Figure 1. The data characteristics of column $T.t_3$ request this column to contain 8 unique strings with average length 20 and maximum length 100. In the result generator, based on the indexes in $[0, 7]$ generated by random index generator, the transformer outputs random strings with the desired lengths, at probabilities $\{p_{0,2,3,5,7} = 0.1, p_1 = 0.2, p_{4,6} = 0.15\}$. The probability table only explicitly specifies the probabilities for 3 indexes, i.e., p_1, p_4 and p_6. The details of probability assignment will be discussed in Section 3.

Value Generation: Given the random column generator, firstly, a *Null* value is output with the probability of the specified percentage. If *Null* value is not chosen, the index generator picks up an index based on the probabilities by running binary search over CDF (c_i) in the probability table with a random real number in $(0, 1]$, and the transformer outputs the corresponding column value.

Specifically, the index generator randomly chooses a real number x in $(0,1]$. By running binary search over CDF (c_i) in the probability table, the index generator retrieves an entry (k_i, p_i, c_i) that its c_i covers the random number x, i.e., $x \leq c_i$ and $x > c_{i-1}$ (or only $x \leq c_i$ when $i = 0$). If $x > c_i - p_i$, the index k_i is fed into transformer. If $x \leq c_i - p_i$, the index generator randomly chooses an index in the range $[k_{i-1}+1, k_i - 1]$ (or $[0, k_i - 1]$ when $i = 0$) for transformer. When the index generator fails to locate such an entry in the table, it picks up an index in $[k_m, n-1]$ for transformer, where k_m is the largest index in the probability table ($k_m = 0$ when there is no entry) and n is the cardinality of unique values specified on the column. With the input index, the transformer outputs its corresponding value.

3 Query Instantiation

There are two general objectives in query instantiation, targeting to 1) construct the random column generators for each non-key column in the tables; and 2) find concrete values for the variable parameters in the queries. We will follow the running example in Figure 1 to elaborate the procedure of query instantiation.

Generally speaking, the query instantiation component is responsible for handling three types of constraints, i.e., $C^{\sigma}_{=}$, $C^{\sigma}_{\neq}$ and $C^{\bowtie}_{\neq}$. Note that the fourth type of constraints $C^{\bowtie}_{=}$ involves matching between primary and foreign keys, which is taken care of by the data generation process at runtime. In Algorithm 1, we list the general workflow of query instantiation. The algorithm iteratively adjusts the data distributions adopted by the random column generators and the concrete values of the variable parameters, in order to meet the constraints as much as possible. The distribution adjustment on the column generator is accomplished by inserting entries in its probability table. In each iteration, the algorithm re-initializes the column generators (line 3) such that there is no entry in the probability table, namely the probabilities of candidate values are uniform. The algorithm then attempts to adjust the column generators in $\bar{G}$ and the concrete values of the variable parameters in queries $\bar{Q}$ (lines 4-14). Specifically, it firstly adjusts the column generators and instantiates the variable parameters based on the equality constraints over filters (lines 4-7). It then follows to revise the variable parameters in the queries in order to meet the non-equality constraints on filter and join operators (lines 8-14). The details of the adjustment on column generators and the parameter instantiation are presented in the following subsections. The algorithm outputs the new (adjusted) generators $\bar{G}$ and the instantiated queries $\bar{Q}$, when the global relative error for all constraints is within the specified threshold θ or the number of iterations reaches its maximum I.

In the rest of the section, we discuss the processing strategies for these three types of constraints respectively.

Algorithm 1 Query instantiation

```
Input: Initial generators $G$, input queries $Q$, error threshold $\theta$
    and maximal iterations $I$
Output: New generators $\bar{G}$ and instantiated queries $\bar{Q}$
 1: Initialize $\bar{Q} \leftarrow Q$
 2: for all iteration $i = 1$ to $I$ do
 3:     Initialize $\bar{G} \leftarrow G$
 4:     for all constraint $c \in C^{\sigma}_{=}$ do
 5:         Adjust the generator in $\bar{G}$ for the column within $c$
 6:         Instantiate the corresponding parameter in $\bar{Q}$
 7:     end for
 8:     for all $c \in C^{\sigma}_{\neq}$ do
 9:         Instantiate the corresponding parameter in $\bar{Q}$
10:     end for
11:     for all $c \in C^{\bowtie}_{\neq}$ do
12:         Obtain constraints from all descendant nodes
13:         Instantiate the corresponding parameter in $\bar{Q}$
14:     end for
15:     Calculate the global relative error $e$
16:     if $e \leq \theta$ then
17:         return $\bar{G}$ and $\bar{Q}$
18:     end if
19: end for
20: return $\bar{G}$ and $\bar{Q}$ (historical best solution with minimum $e$)
```

Filters with Equality Constraint always involve a single non-key column at a time like the workloads of standard benchmarks. Given all these equality constraints on the filter operators, i.e., $C^{\sigma}_{=}$, the system groups the constraints according to the involved column. In our running example in Figure 1, there are four such constraints $C^{\sigma}_{=} = \{c_2, c_5, c_8, c_{10}\}$, among which, c_2 and c_8 target column $S.s_3$, and c_5 and c_{10} target column $T.t_3$. Note that all relational operators in equality constraints are handled by treating them as '='. For example, $c_5 = [Q_2, T.t_3 \text{ NOT LIKE } P_4, 32] \Rightarrow [Q_2, T.t_3 \text{ LIKE } P_4, 50 \times (1 - 20\%) - 32] \Rightarrow [Q_2, T.t_3 = P_4, 8]$, where 20% is the percentage of *Null* value in column $T.t_3$.

Based on the workflow in Algorithm 1, in each iteration, the query instantiation algorithm always reinitializes the random column generators by the uniform distribution over the candidate values for every column. The processing strategy for equality constraints on filters runs in three steps. Firstly, the algorithm randomly selects an index and obtains the corresponding value from the transformer of the column generator, for instantiating each variable parameter in the equality constraints. Secondly, the algorithm updates the occurrence probability of the selected index in the column generator by inserting an entry in the probability table, in order to meet the required intermediate result cardinality. Whether the filter is the leaf node of the query execution tree or not, the probability of the inserted entry is calculated as $\frac{s_{out}}{s_{in}}$, where s_{in} is the size of input tuples and s_{out} is the ex-

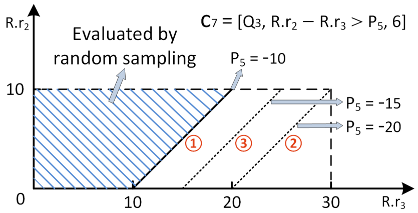


Figure 4: An example of parameter searching procedure for constraint c_7 in our running example. Given the predicate in the constraint, our algorithm attempts to cut the space by revising the parameter P_5. For a concrete P_5, the expected number of tuples meeting the predicate is evaluated by the random sampling algorithm. The best value for P_5 is returned, after the binary search identifies the optimal value at desired precision or reaches the maximum iterations.

pected size of output tuples. After the above two steps for all equality constraints, the algorithm calculates the cumulative probabilities in the probability table of adjusted column generators. In Figure 3, there are three entries in the probability table for generating data with the distribution that satisfies the constraints c_5 and c_{10}. For example, the entry with index 1 is inserted for instantiating parameter P_4 in the predicate of c_5, while the two entries with indexes 4 and 6 are inserted for instantiating parameters P_7 and P_8 in the predicate of c_{10}.

Suppose there are k variable parameters in the equality constraints over filters. The total complexity of the processing strategy is $O(k \log k)$, because the algorithm only needs to instantiate the parameters one by one, and accordingly it inserts an entry into the probability table in order of selected index for each parameter instantiation.

Filters with Non-Equality Constraint could involve multiple non-key columns, and the values of involved columns require satisfying the specified non-equality relationship. In Figure 1, some constraints, e.g., $c_1 = [Q_1, R.r_2 < P_1, 4]$ and $c_4 = [Q_2, S.s_3 \geq P_3, 7]$, apply on one column only, while other constraints, e.g., $c_7 = [Q_3, R.r_2 - R.r_3 > P_5, 6]$ and $c_{12} = [Q_4, 2 \times R.r_2 + R.r_3 < P_9, 7]$, involve more than one column with more complex mathematical operations. Our underlying strategy handling these non-equality constraints is to find the concrete parameters generating the best matching output cardinalities against the constraints, based on the data distributions adopted by the random column generators.

Since the cardinality of tuples satisfying the constraints is monotonic with the growth of the variable parameter, it suffices to run a binary search over the parameter domain to find the optimal concrete value for the variable parameter. In Figure 4, we present an example to illustrate the parameter searching procedure. The cutting line in the figure represents the parameter in the constraint, which decides the ratio of tuples in the shadow area, i.e., satisfying the constraint. By increasing or decreasing the parameter, the likelihood of tuples in the shadow area changes correspondingly. The technical challenge behind the search is the hardness of likelihood evaluation over the satisfying tuples, or equivalently the probability of tuples falling in the shadow area in our example. To tackle the problem, we adopt the random sampling algorithm, which is also suited for the non-uniform distribution of the involved columns. Note that the binary search may not be able to find a parameter with the desired precision, based on the determined data distribution of columns after processing equality constraints over filters. Therefore, in Algorithm 1, we try to instantiate the parameters for non-equality constraints upon different data distributions by iteration. Since the sizes in equality constraints over filters are usually far less than table sizes, the processing of $C^{\sigma}_{=}$ does not make the data distributions of column values extremely skewed. Hence, Algorithm 1 can often achieve the query instantiation through only one iteration in our experiments while satisfying the error requirement.

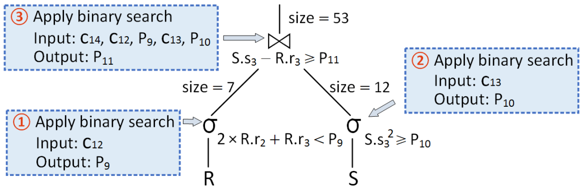


Figure 5: An example of parameter instantiation for non-equality constraints on join operator

The complexity of the approach is the product of two components, the number of iterations in parameter value search and the computational cost of probability evaluation using random sampling algorithm in each iteration. The number of iterations for the binary search is logarithmic to the domain size of the parameter, decided by the minimal and maximal value that the expression with multiple columns could reach. The cost of random sampling depends on the complexity of the predicate, which usually only involves a few columns.

Joins with Non-Equality Constraint is slightly different from the filters with non-equality constraints, because the columns involved in their predicates may overlap with the columns in the predicates of their child nodes as query Q_4 in Figure 1, which usually does not happen to filters in the query execution tree. Therefore, we must process the constraints in a bottom-up manner without the premise of probability independence, such that the precedent operators are settled before the join operator with non-equality constraint is handled. In Figure 5, we present the procedure on query Q_4. After *Touchstone* concretizes the parameters P_9 and P_{10} in constraints c_{12}

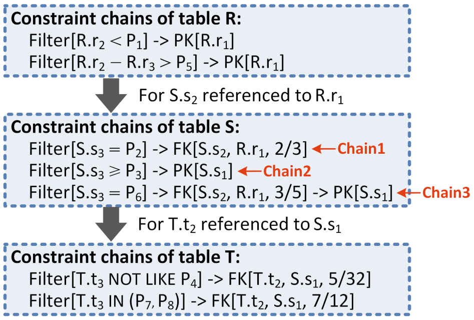

Constraint chains of table R:
Filter[$R.r_2 < P_1$] -> PK[$R.r_1$]
Filter[$R.r_2 - R.r_3 > P_5$] -> PK[$R.r_1$]
For $S.s_2$ referenced to $R.r_1$
Constraint chains of table S:
Filter[$S.s_3 = P_2$] -> FK[$S.s_2$, $R.r_1$, 2/3]
Chain1
Filter[$S.s_3 \geqslant P_3$] -> PK[$S.s_1$]
Chain2
Filter[$S.s_3 = P_6$] -> FK[$S.s_2$, $R.r_1$, 3/5] -> PK[$S.s_1$]
Chain3
For $T.t_2$ referenced to $S.s_1$
Constraint chains of table T:
Filter[$T.t_3$ NOT LIKE P_4] -> FK[$T.t_2$, $S.s_1$, 5/32]
Filter[$T.t_3$ IN (P_7, P_8)] -> FK[$T.t_2$, $S.s_1$, 7/12]

Figure 6: Results of constraint chain decomposition

and c_{13}, the input data to the join operator with constraint c_{14} are determined. Based on the characteristics of the inputs, we apply the same binary search strategy designed for filter operator to construct the optimal parameter, e.g., P_{11} in Figure 5, for the desired result cardinality. Since the algorithm is identical to that for filter operator, we hereby skip detailed algorithm descriptions as well as the complexity analysis.

4 Data Generation

Given the generators of all non-key columns and the instantiated queries, the data generation component in Figure 2 is responsible for assembling tuples based on the outputs of the column generators. The key technical challenge here is to meet the equality constraints over the join operators, i.e., $C^{\bowtie}_{=}$, which involve the dependencies among primary and foreign keys from multiple tables. To tackle the problem, we design a new tuple generation schema, which focuses on the manipulation of foreign keys only.

The tuple generation consists of two steps. In the first *compilation* step, *Touchstone* orders the tables as a generation sequence and decomposes the query trees into constraint chains for each target table. In the second *assembling* step, the working threads in *Touchstone* independently generate tuples for the tables based on the result order from compilation step. For each tuple, the working thread fills values in the columns by calling the random column generators independently and incrementally assigns a primary key, while leaving the foreign keys blank. By iterating the constraint chains associated with the table, the algorithm identifies the appropriate candidate keys for each foreign key based on the maintained join information of the referenced primary key, and randomly assigns one of the candidate keys to the tuple.

Compilation Step: The generation order of the tables is supposed to be consistent with the dependencies between primary keys and foreign keys, because the primary key must be generated before the adoption of its join information for generating corresponding foreign keys of other tables. Since such primary-foreign-key dependencies form a directed acyclic graph (DAG), *Touchstone* easily constructs a topological order over the tables. In Figure 6, we illustrate the result order over three tables, $R \rightarrow S \rightarrow T$, based on the database schema H in Figure 1.

In order to decouple the dependencies among columns and facilitate parallelizing, *Touchstone* decomposes the query trees annotated with constraints into *constraint chains*. A constraint chain consists of a number of constraints corresponding to the cardinality constraints over the operators in query trees. There are three types of constraints included in the constraint chains, namely FILTER, PK and FK, which are associated with the types of related operators. The constraint chains with respect to a table are defined as the sequences of constraints with descendant relationship in the query trees. In Figure 6, we present all the constraint chains for tables R, S and T. Table R has two constraint chains extracted from query Q_1 and Q_3. And the constraint chains of table S are marked in Figure 1 for easily understanding. Note that there is no constraint chain from query Q_4, because there is no join operator with the equality constraint in Q_4.

Each FILTER constraint keeps the predicate with the instantiated parameters. Each PK constraint in the chain records the column name of the primary key. Each FK constraint maintains a triplet, covering two column names of the foreign key and the referenced primary key, and the expected *ratio* of tuples satisfying the predicate on the join operator. The second constraint in the first chain for table S in Figure 6, for example, is FK$[S.s_2, R.r_1, \frac{2}{3}]$, indicating the foreign key is $S.s_2$, the referenced primary key is $R.r_1$ and two out of three tuples in table S are expected to meet the predicate $S.s_2 = R.r_1$ of join operator in the case of satisfying the predicate $S.s_3 = P_2$ of previous filter. The expected ratios in FK constraints are calculated based on the cardinality requirements of the specified cardinality constraints.

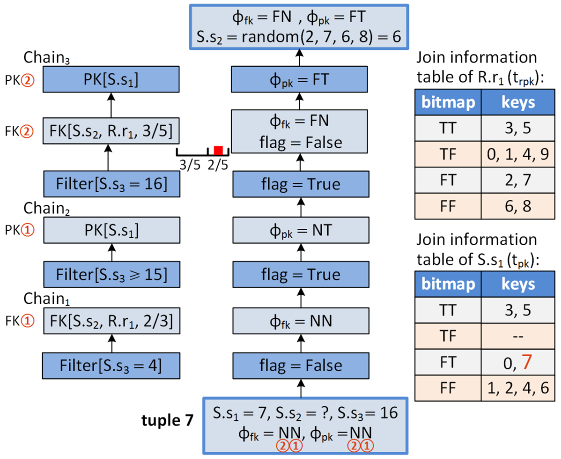
Assembling Step: For simplicity, we assume that there is a single-column primary key and one foreign key in the table. Note that our algorithm can be naturally extended to handle tables with composite primary key and multiple foreign keys. The result constraint chains are distributed to all working threads on multiple nodes for parallel tuple generation. When generating tuples for a specified table, each working thread maintains two bitmap data structures at runtime, i.e., ϕ_{fk} and ϕ_{pk}. They are used to keep track of the status of joinability, e.g., whether the generating tuple satisfies individual predicates over join operators, for primary key and foreign key, respectively. The length of the bitmap ϕ_{fk} (resp. ϕ_{pk}) is equivalent to the number of FK (resp. PK) constraints in all chains of the target table. Each bit in the bitmap corresponds to a FK/PK constraint. It has three possible values, T, F

and N, indicating if the join status is successful, unsuccessful or null. In Figure 6, for example, table S has two FK constraints and two PK constraints, resulting in 2-bit representations for both ϕ_{fk} and ϕ_{pk}.

Touchstone also maintains the join information table to track the status of joinability of primary keys based on the bitmap representation ϕ_{pk}. In Figure 7, we show two join information tables of primary keys $R.r_1$ and $S.s_1$ respectively. The join information table of $R.r_1$ is maintained in the generation of table R, which is ready for generating the foreign key $S.s_2$ of table S. During the generation of table S, the join information table of $S.s_1$ is maintained for generating the foreign key $T.t_2$ of table T. There are two attributes in the entry of join information table, i.e., bitmap and keys, indicating the status of joinability and the corresponding satisfying primary key values. Note that the keys in the entry may be empty (such entries will not be stored in practice), which means there is no primary key with the desired joinability status.

Algorithm 2 Tuple generation

```
Input: Column generators $\bar{G}$, constraint chains of the target
    table $\Omega$, join information tables of referenced primary key
    and current primary key $t_{rpk}$ and $t_{pk}$
Output: Tuple $r$ and join information table $t_{pk}$
 1: $r.pk \leftarrow$ a value assigned incrementally
 2: $r.columns \leftarrow$ values output by column generators $\bar{G}$
 3: $\phi_{fk} \leftarrow N...N$, $\phi_{pk} \leftarrow N...N$
 4: for all constraint chain $\omega \in \Omega$ do
 5:     $flag \leftarrow True$
 6:     for all constraint $c \in \omega$ do
 7:         if ($c$ is FILTER) && ($c.predicate$ is $False$) then
 8:             $flag \leftarrow False$
 9:         else if $c$ is PK then
10:             $\phi_{pk}[i] \leftarrow flag$   // $i$ is the bit index for $c$
11:         else if ($c$ is FK) && $flag$ then
12:             if $random[0,1) \geq c.ratio$ then $flag \leftarrow False$.
13:             $\phi_{fk}[i] \leftarrow flag$   // $i$ is the bit index for $c$
14:         end if
15:     end for
16: end for
17: $r.fk \leftarrow$ a value selected from $t_{rpk}$ satisfying $\phi_{fk}$
18: Add $r.pk$ in the entry of $t_{pk}$ with bitmap $\phi_{pk}$
19: return  $r$ and $t_{pk}$
```



Join information table of R.r_1 (t_{rpk}):

bitmap	keys
TT	3, 5
TF	0, 1, 4, 9
FT	2, 7
FF	6, 8

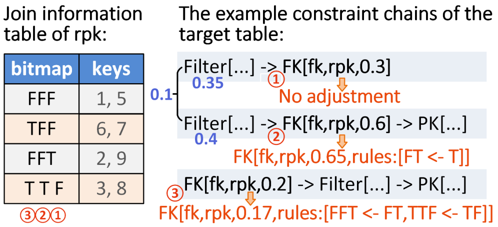
Join information table of S.s_1 (t_{pk}):

bitmap	keys
TT	3, 5
TF	--
FT	0, 7
FF	1, 2, 4, 6

Figure 7: Running example of tuple generation for table S

The tuple generation algorithm is listed in Algorithm 2. We present a running example of tuple generation in Figure 7. A new tuple for table S is initialized as ($S.s_1 = 7, S.s_2 = ?, S.s_3 = 16$), $\phi_{fk} = NN$ and $\phi_{pk} = NN$ (lines 1-3). The $flag$ is set to True before traversing each constraint chain (line 5), which is used to track if the predicates from the precedent constraints of current chain are fully met. On the first constraint chain, since the predicate in the first FILTER constraint is $S.s_3 = 4$, $flag$ is then set to False (line 8), and algorithm does not need to handle the next FK constraint (line 11). On the second chain, the tuple satisfies the predicate $S.s_3 \geq 15$, resulting in the update of bitmap representation as $\phi_{pk} = NT$ (line 10). On the third chain, after passing the first FILTER constraint, the corresponding bit of next FK constraint in ϕ_{fk} is randomly flipped to F at the probability of $\frac{2}{5}$ (lines 12-13), because the expected *ratio* of satisfying tuples is $\frac{3}{5}$. The $flag$ is set to False (line 12) to reflect the failure of full matching of precedent constraints for later PK constraint. Then, the bit corresponding to next PK constraint in ϕ_{pk} is set as F according to the value of $flag$ (line 10). Therefore, the two bitmaps are finalized as $\phi_{fk} = FN$ and $\phi_{pk} = FT$. Then the algorithm identifies (line 17) two entries matching $\phi_{fk} = FN$, namely satisfying the T/F requirements on the corresponding bits of ϕ_{fk}, with bitmaps FT and FF respectively, in the join information table of $R.r_1$. Given these two entries, it randomly selects (line 17) a foreign key, e.g., 6, from four candidate referenced primary keys $\{2, 7, 6, 8\}$, which are all appropriate as the foreign key $S.s_2$. That there is no entry in t_{rpk} satisfying the T/F requirements of ϕ_{fk}, which is called *mismatch case*, is dealt in the rest of the section. Finally, the algorithm updates (line 18) the join information table of $S.s_1$ by adding the primary key $S.s_1 = 7$ into the entry with bitmap FT.

For a table, suppose there are k non-key columns, m constraints in the related constraint chains and n entries in the join information table of referenced primary key. The complexity of tuple generation mainly consists of three parts, k times of calling random column generators for filling the values of non-key columns, the traversing over m constraints within chains for determining the joinability statuses of foreign key and primary key, and the comparing with n bitmaps in the join information table for searching the appropriate foreign key candidates. For practical workloads, k, m and n are all small numbers,


Join information table of rpk:
The example constraint chains of the target table:
Filter[...] -> FK[fk,rpk,0.3]
0.35
0.1
①
No adjustment
Filter[...] -> FK[fk,rpk,0.6] -> PK[...]
0.4
②
FK[fk,rpk,0.65,rules:[FT <- T]]
FK[fk,rpk,0.2] -> Filter[...] -> PK[...]
③
FK[fk,rpk,0.17,rules:[FFT <- FT,TTF <- TF]]
③②①

bitmap	keys
FFF	1, 5
TFF	6, 7
FFT	2, 9
T T F	3, 8

Figure 8: An example of adjustments to FK constraints

e.g., $k \leq 12$, $m \leq 20$ and $n \leq 40$ for TPC-H [3] workload, so our tuple generation is highly efficient.

Handling Mismatch Cases: For the data generation of big tables, if a joinability status of the primary key may occur, its occurrence can be considered as inevitable based on the probability theory. However, there are still some joinability statuses of the primary key that never occur. For example, in Figure 7, the bitmap ϕ_{pk} for primary key $S.s_1$ can not be TF due to the constraints, i.e., Filter[$S.s_3 = 16$] and Filter[$S.s_3 \geq 15$]. Therefore, in the tuple generation, it should be avoided to generate the bitmap ϕ_{fk} that does not have any matching entry in the join information table of the referenced primary key. In order to achieve this objective, the main idea is to add rules to manipulate relevant FK constraints.

Figure 8 gives an example of adjustments to FK constraints for handling the mismatch case. There are three FK constraints with the serial numbers of 1, 2 and 3 in the three constraint chains, respectively. Since there are four bitmaps, i.e, FTT, TTT, TFT, FTF, that are not present in the join information table of the referenced primary key rpk corresponding to the foreign key fk of the target table, three rules are added in two FK constraints to avoid producing any ϕ_{fk} triggering the mismatch case. For example, there is a rule $[FT \leftarrow T]$ added in the second FK constraint, which indicates that the status of the second FK constraint must be F if the status of the first FK constraint is T in the tuple generation. Since there are extra F statuses forcibly generated by the added rule for the second FK constraint, the actual ratio of tuples satisfying the corresponding predicate could be lower than the expected ratio 0.6. Consequently, it is necessary to adjust the ratio in the second FK constraint for eliminating the impact of the added rule. In this example, we adjust the ratio as $0.65 = \frac{0.6 \times 0.4}{0.4 - 0.1 \times 0.3}$, in which 0.4 is the ratio of tuples satisfying the predicate in the second FILTER constraint, 0.6×0.4 is the cumulative probability of the status T for the second FK constraint, 0.1 is the ratio of tuples satisfying the two predicates in the first two FILTER constraints, 0.3 is the ratio in the first FK constraint and 0.1×0.3 is the cumulative probability of the extra F status generated by the rule.

Algorithm 3 Adjustments to FK constraints

Input: Constraint chains Ω and join information table of referenced primary key t_{rpk}

Output: Adjusted constraint chains $\bar{\Omega}$

```
1:  for all $\bar{\Omega}$ initialized from $\Omega$ in a random order do
2:      Initialize $C \leftarrow$ all FK constraints ordered as in $\bar{\Omega}$
3:      for all index $i = 1$ to $C.size$ do
4:          for all possible joinability status $\phi$ of $C_1$ to $C_i$ do
5:              if mismatch case is occurred over $\phi$ and $t_{rpk}$ then
6:                  Add a rule built according to $\phi$ into $C_i$
7:              end if
8:          end for
9:          Adjust $C_i.ratio$ based on added rules
10:     end for
11:     Calculate error $e$ caused by illegal adjustments
12:     if $e = 0$ then
13:         return $\bar{\Omega}$
14:     end if
15: end for
16: return $\bar{\Omega}$ (historical best solution with minimum $e$)
```

In Algorithm 3, we list the general workflow of adjustments to FK constraints. Note that the algorithm is called only once before starting the tuple generation for the target table. The adjustment is based on the join information tables of referenced primary keys, so the referenced tables must have been generated before. We assume that there is one foreign key in the table, and the adjustment can be done separately over the related FK constraints for each foreign key if there are multiple foreign keys. Firstly, the algorithm initializes constraint chains $\bar{\Omega}$ from Ω in a random order (line 1). Please note a poorly chosen order may not generate desired adjustment results. In the worst case, the algorithm may need to attempt all possible orders of the constraint chains, behind the NP-hardness nature of the problem. Given the specified order of the constraint chains (line 2), the algorithm then traverses all FK constraints in order (line 3). For i-th FK constraint, i.e., C_i, the algorithm iterates over all possible joinability statuses considering constraints C_1 to C_i to identify which joinability statuses may generate mismatch cases (lines 4-8). For example, when it reaches the third FK constraint, the algorithm iterates over 2^3 joinability statuses $\{s_3 s_2 s_1\}$ with $s_i \in \{T, F\}$. For any $s_i = N$, since it indicates there is no requirement on the joinability status for the referenced primary key and never triggers any mismatch case, it is unnecessary to check it. When a mismatch case is detected (line 5) over a joinability status ϕ, namely there is no satisfying referenced primary key in t_{rpk}, a corresponding rule is added into current FK constraint (line 6). For example, if the mismatch case occurs over $\phi = TFT$, a rule $[FFT \leftarrow FT]$ is generated for the third FK constraint, indicating that its status s_3 must be F if the statuses of previous two FK constraints $s_2 s_1 = FT$. After iterating over all possible joinability statuses for i-th FK constraint, the algorithm adjusts its ratio based on the added rules to ensure the expected output cardinality (line 9). Next, when all FK constraints in constraint chains $\bar{\Omega}$ are processed, the algorithm calculates the error caused by all illegal adjustments that the corresponding adjusted ratios are not between 0 and 1 (line 11). The detailed calculation of above ratio adjustment and error can be found in Appendix A. Finally, the algorithm outputs the constraint chains $\bar{\Omega}$ with adjusted FK constraints, when there is no illegal adjustments or it has traversed all the orders of constraint chains.

The constraint chains are decomposed from the instantiated query trees, and the primary-foreign-key constraints are also specified in the constraint chains. Therefore, the constraint chains and the query trees are equivalent in the expression of cardinality constraints. Thus, if the cardinality constraints $C^{\bowtie}_{=}$ can be well satisfied on the current instantiated query trees, there must be constraint chains in a certain order that the error caused by illegal adjustments is acceptable, namely the result generated data can well satisfy $C^{\bowtie}_{=}$. But the illegal inputs to the *Touchstone* system may lead to a big error. Specially, for a large set of constraint chains, after k rounds of attempts without iterating all possible orders, we have a solution among the top 1% smallest-error ones with the probability of $p = 1-(1-1\%)^k$, where $p > 99.99\%$ with $k = 10^3$.

To reflect the adjustments to FK constraints in tuple generation, minor modification is applied on the original tuple generation algorithm on lines 12-13 in Algorithm 2. Specifically, the updated algorithm first checks all existing rules in current FK constraint. If there is a rule which can be applied to the statuses of previous constraints, ϕ_{fk} and $flag$ are updated according to the rule. Otherwise, the algorithm updates ϕ_{fk} and $flag$ by the probability based on the adjusted ratio. In addition, it would not have two conflicting rules, e.g., rules $[FFT \leftarrow FT]$ and $[TFT \leftarrow FT]$, applied at the same time for any FK constraint. All conflicting rules in the FK constraints would never be applied, and therefore are removed for reducing the complexity of tuple generation. The detailed discussion about the conflicting rules is presented in Appendix B.

For a table with one foreign key, suppose there are k FK constraints in all chains and m entries in the join information table of the referenced primary key. We only need to consider the order of those k constraint chains which include the FK constraint. The complexity of the adjustment is $O(k! \cdot (\sum_{1 \leq i \leq k} 2^i) \cdot m)$, where $k!$ is the number of all possible orders of constraint chains (or FK constraints), $\sum_{1 \leq i \leq k} 2^i$ is the sum of numbers of all possible joinability statuses for processing all FK constraints and m is the times of checking whether the mismatch case is occurred over a joinability status. For practical workloads, k and m are both small numbers, e.g., $k \leq 8$ and $m \leq 40$ for TPC-H workload, and the adjustment is only done once for a table. Therefore, the cost of such adjustment is usually affordable. To save unnecessary computation, we could find a solution among the best top 1% ones by running a constant number of iterations, basically replacing $k!$ with a constant number in the complexity analysis. The complexity of tuple generation is increased by n times of rule checking, where n is the number of rules added in FK constraints, and usually small for practical workloads, e.g., $n \leq 41$ for TPC-H workload.

Management of Join Information: For generation of a table, it can be completely parallel on multiple nodes with multiple working threads on each node. Each working thread maintains its own join information table of the primary key to avoid contention. But the join information table of referenced primary key can be shared among multiple working threads on each node. After the generation of the table, we merge the join information tables maintained by the multiple working threads in distributed controller as in Figure 2. But there are serious memory and network problems for the space complexity of the join information table is $O(s)$ with s as the table size.

Since the relationship of foreign key and primary key can be many to one and the intermediate result cardinality is the main factor that affects the query performance, we design a compression method by storing less primary key values in the join information table but still promise the randomness of remaining values. Assuming the size of keys in an entry of join information table is N, which is hard to know in advance and may be very large. We aim to store only L ($L << N$) values in the keys and promise the approximate uniform distribution of these L ones among all N values. The compression method is implemented as follows: we store the first L arriving values in the keys, if any; and for the i-th ($i > L$) arriving value, we randomly replace a value stored previously in the keys with the probability of L/i. By such a method, the space complexity of the join information table is reduced to $O(n * L)$, where n is the number of entries in the join information table and L is the maximum allowed size of keys in each entry. Since n is generally small, e.g., $n \leq 40$ for TPC-H workload, and L usually can be set to thousands, the memory consumption and network transmission of the join information table are acceptable.

5 Experiments

Environment. Our experiments are conducted on a cluster with 8 nodes. Each node is equipped with 2 Intel Xeon E5-2620 @ 2.0 GHz CPUs, 64GB memory and 3

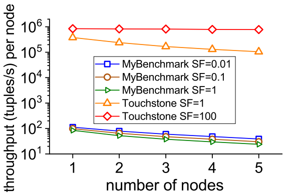


Figure 9: Comparison of data generation throughput

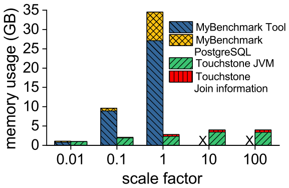


Figure 10: Comparison of memory consumption

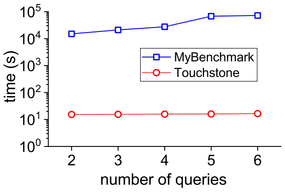


Figure 11: Comparison of data generation time

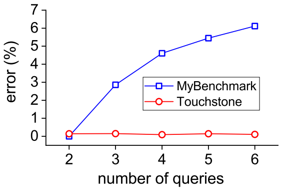


Figure 12: Comparison of global relative error

TB HDD disk configured in RAID-5. The cluster is connected using 1 Gigabit Ethernet.

Workloads. The TPC-H [3] is a decision support benchmark which contains the most representative queries of analytical applications, while the transactional benchmarks, e.g., TPC-C and TPC-W, do not contain queries for analytical processing. So we take the TPC-H workload for our experiments. We compare *Touchstone* with the state-of-the-art work MyBenchmark [15] with source codes from the authors.[2] The workloads for comparison consist of 6 queries from TPC-H, including $Q_{2,3,6,10,14,16}$. Note that these queries are selected based on the performance of MyBenchmark, which drops significantly when other queries are included in the workloads. *Touchstone*, on the other hand, can easily handle all of the first 16 queries, i.e., Q_1 to Q_{16}, in TPC-H with excellent performance. To the best of our knowledge, *Touchstone* provides the widest support to TPC-H workload, among all the existing studies [6, 14, 5, 15].

Input Generation. To build valid inputs for experiments, we generate the DBI and queries of TPC-H using its tools *dbgen* and *qgen*, respectively. And the DBI of TPC-H is imported into the MySQL database. The database schema of TPC-H is used as the input H. We can easily obtain the input data characteristics D for all columns from the DBI in MySQL. Given the TPC-H queries, their physical query plans are obtained from MySQL query parser and optimizer over the DBI. The cardinality constraints corresponding to the operators in query plans are then identified by running the queries on the DBI in MySQL. The input workload characteristics W are constructed by the parameterized TPC-H queries and above cardinality constraints. Note that we can generate databases with different scale factors using the same input W by employing selectivities instead of the absolute cardinalities in our input constraints.

Settings. As data is randomly generated according to the column generators in *Touchstone*, the distribution of generated data may be difficult to satisfy the expectation for small tables such as *Region* and *Nation*. We therefore revise the sizes of *Region* and *Nation* from 5 to 500, and from 25 to 2500 respectively. The cardinality constraints involving these two tables are updated proportionally. In addition, the small tables can also be pre-generated manually. The error threshold (desired precision) and maximal iterations in query instantiation are set to 10^{-4} and 20 respectively. The default maximum allowed size L of keys in join information table is set to 10^4.

5.1 Comparison with MyBenchmark

We compare *Touchstone* with MyBenchmark from four aspects, including data generation throughput, scalability to multiple nodes, memory consumption and capability of complex workloads.

Figure 9 shows the data generation throughputs per node of *Touchstone* and MyBenchmark as we vary the number of nodes under different scale factors. Due to the unacceptably long processing time of MyBenchmark, we adopt smaller scale factors for it and large scale factors for *Touchstone*. Overall, the data generation throughput of *Touchstone* is at least 3 orders of magnitude higher than that of MyBenchmark. This is because MyBenchmark does not have a good parallelization or an efficient data generation schema. Furthermore, as the number of nodes increases from 1 to 5, the data generation throughput per node of MyBenchmark decreases dramatically for all three scale factors. Although the decline of data generation throughput per node of *Touchstone* is obvious too when $SF = 1$, *Touchstone* is linearly scalable (the throughput per node is stable) when $SF = 100$. This is because for small target database, e.g., $SF = 1$, the distributed maintenance rather than data generation dominates the computational cost in *Touchstone*, while its overhead comparatively diminishes by increasing the target database size.

Figure 10 reports the peak memory consumptions of *Touchstone* and MyBenchmark under different data scales. The experiment is conducted on 5 nodes with no restriction on memory usage. The memory usage of MyBenchmark mainly consists of two parts, namely, memory consumed by MyBenchmark Tool and memory consumed by PostgreSQL for managing intermediate states. The memory usage of *Touchstone* mainly in-

[2] We would like to thank Eric Lo for providing us the source code of MyBenchmark.

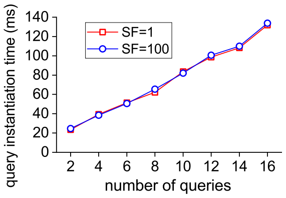


Figure 13: Query instantiation time

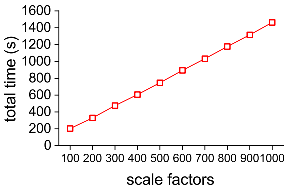


Figure 14: Total running time

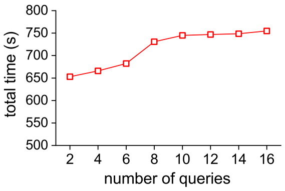


Figure 15: Scalability to data scale

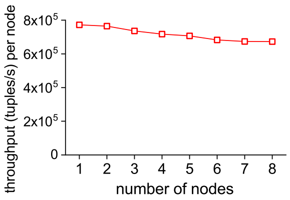


Figure 16: Scalability to multiple nodes

cludes memory for JVM itself and memory for maintaining join information. As shown in Figure 10, the memory consumption of *Touchstone* is much lower than that of MyBenchmark under the same scale factors. This is because the symbolic databases and constrained bipartite graphs in MyBenchmark are very large and consume a large amount of memory, but the join information table in *Touchstone* has been effectively compressed. It is worth noting that the memory consumption of *Touchstone* remains almost constant when $SF > 10$. This is because for *Touchstone*, the JVM itself occupies most of the memory, while the join information maintenance only spends a tiny piece of memory.

Figure 11 and Figure 12 present the data generation time (total running time) and global relative error separately of *Touchstone* and MyBenchmark as we vary the number of input queries with $SF = 1$. The input queries are loaded in order of their serial numbers. The experiment is carried out on 5 nodes. In Figure 11, it is obvious that the data generation time of MyBenchmark increases significantly as the number of queries increases. At the same time, the generation time of *Touchstone* grows very little when more queries are included, significantly outperforming MyBenchmark. In Figure 12, the error of *Touchstone* is much smaller than that of MyBenchmark. Moreover, as there are more input queries, the global relative error of *Touchstone* remains small with little change, while the error of MyBenchmark has an obvious rise. In summary, *Touchstone* is more capable of supporting complex workloads than MyBenchmark.

It can be seen from previous experiments that MyBenchmark can not be easily applied to generate the terabyte scale database for complex workloads due to its poor performance. In the following, we further demonstrate the advantages of *Touchstone* by a series of experiments using the workload of 16 queries, i.e., Q_1 to Q_{16}.

5.2 Performance Evaluation

In this section, we evaluate the impact of workload complexity on query instantiation time and total running time in *Touchstone*, as well as the scalability to data scale and multiple nodes of *Touchstone*.

Figure 13 shows the query instantiation time of *Touchstone* as we vary the number of queries with $SF = 1$ and $SF = 100$, respectively. The input queries are loaded in order of their serial numbers. The query instantiator is deployed on a single node. As shown in Figure 13, even when all 16 queries are used for input, query instantiation is finished within 0.2s. And there is a minimal difference in query instantiation time for $SF = 1$ and $SF = 100$, as the complexity of query instantiation is independent of data scale. Overall, the query instantiation time is only correlated to the complexity of input workloads.

Figure 14 shows the total running time of *Touchstone* as we vary the number of queries with $SF = 500$. *Touchstone* is deployed on 8 nodes. From the result, it can be seen that the running time increases slowly as the number of queries increases. For Q_7 and Q_8, there are relatively more cardinality constraints over equi-join operators, so the time increment is larger when we change from 6 queries to 8 queries. But when the number of queries changed from 10 to 16, the time increment is almost indiscernible, for Q_{11} to Q_{16} are simple, among which Q_{12} to Q_{15} have no cardinality constraints on equi-join operators[3]. Overall, the total running time increased by only 16% from 2 queries to 16 queries for 500GB data generation task, so *Touchstone* is insensitive to the workload complexity.

Figure 15 presents the total running time of *Touchstone* under different scale factors with the input of 16 queries. *Touchstone* is deployed on 8 nodes. As shown in Figure 15, *Touchstone* is linearly scalable to data size. Because the generation of each tuple is independent and the generated tuples need not be stored in memory, the data generation throughput is stable for different data scales. For $SF = 100$ and $SF = 1000$ in Figure 15, the total runtime is increased from 200s to 1460s (less than ten times). When scale factor increases by 100, the runtime increases about 140s, which is less than the total runtime of $SF = 100$. Since we are generating data in a distributed environment, the extra cost is about 60s in current deployment, which includes merging, compress-

[3] Depending on the physical query plans of Q_{12} to Q_{15}, the primary keys in their equi-join operators are from the original tables, so all foreign keys must be joined and the sizes of output tuples are determined.

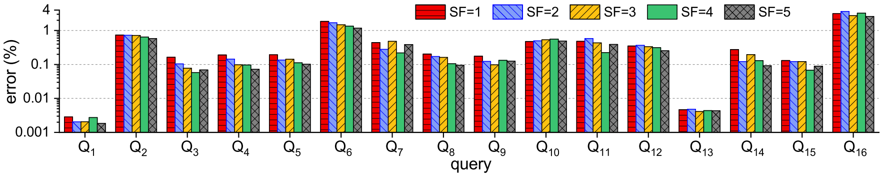


Figure 17: Relative error for each query

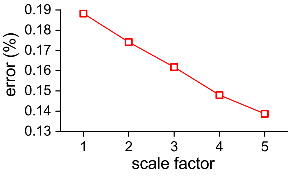


Figure 18: Global relative error vs. scale factor

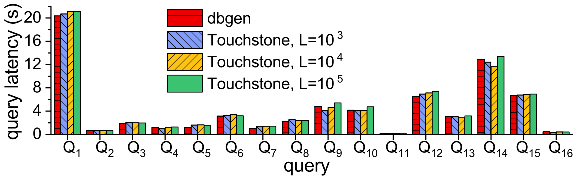


Figure 19: Performance deviation for each query

ing and network transmission of join information tables. And the extra cost is almost the same for different scale factors. Moreover, the total runtime of *Touchstone* is less than 25 minutes for $SF = 1000$ (1TB), so it is capable of supporting industrial scale database generation.

Figure 16 presents the data generation throughputs per node of *Touchstone* as we vary the number of nodes with $SF = 500$. The input workload includes 16 queries. The distributed controller is deployed on one node, and then we continually add nodes to deploy the data generator. The result shows that the data generation throughput per node is approximatively unchanged as the number of nodes increases, validating the linear scalability of *Touchstone*. To the best of our knowledge, *Touchstone* is the first query-aware data generator which can support full parallel data generation on multiple nodes.

5.3 Data Fidelity Evaluation

The data fidelity of synthetic database is evaluated by relative error on cardinality constraints and performance deviation on query latencies. We calculate the relative error for each query in the similar way with global relative error, which only involves its own cardinality constraints. We compare the latency of query processing on base database generated by *dbgen* against that on synthetic database generated by *Touchstone* to show the performance deviation.

Figure 17 shows the relative errors for Q_1 to Q_{16} with different scale factors from 1 to 5. Since the cardinalities of each intermediate result in the query tree need to be counted by the database, and MySQL is very slow when importing large sized tables, we run this group of experiments with the scale factors no more than 5. The maximum error among all 16 queries is less than 4%, and there are 14 queries with errors less than 1%. Figure 18 shows the global relative error of all 16 queries as we vary the scale factor, which is less than 0.2% for all scale factors. And with the increase of scale factor, the global relative error has a sharp decrease. Since data is randomly generated by column generators, as expected by the probability theory, the larger the data size, the smaller the relative error.

Figure 19 presents the performance deviations of all 16 queries with $SF = 1$. We vary the maximum allowed size L of keys in the join information table from 10^3 to 10^5. We can see that the performance deviation is inconspicuous for all 16 queries, and the size of L has no significant influence on query latencies. The result strongly illustrates the correctness and usefulness of our work. We are the first work to give such an experiment to verify the fidelity of the generated DBI.

5.4 Support to Non-Equi-Join Workload

Sample workload:
Tables:
TrafLight (10,000): TL_Key, TL_Lng, TL_Lat.
Accident (100,000): A_Key, A_Lng, A_Lat.
Query:
SELECT TL_key, COUNT(*) as Num
FROM TrafLight, Accident
WHERE DISTANCE(TL_Lng, TL_Lat, A_Lng, A_Lat) $\leq$ Para
GROUP BY TL_Key
ORDER BY Num DESC.

We give a sample workload from real world application for non-equi-join operator. There are two tables named as *TrafLight* and *Accident*, which store the traffic light and road accident information respectively. The table schemas are simplified, as we removed the columns that are not relevant to the sample workload, where both tables only contain the primary key and location information. The query returns traffic lights in descending order of the number of road accidents happened nearby, and there is a non-equi-join operator between table *TrafLight* and table *Accident*. So that we can know which traffic junctions are more likely to have road accidents, and then

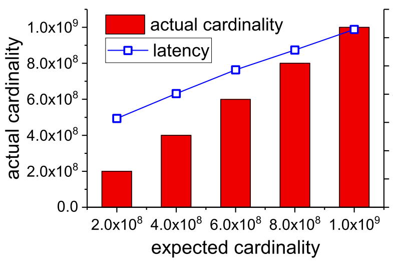


Figure 20: Data generation for non-equi-join workload

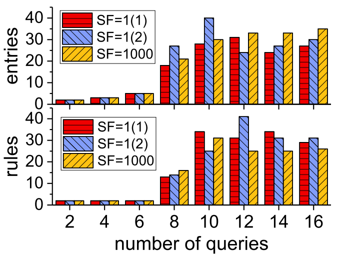


Figure 21: The maximum numbers of entries and rules

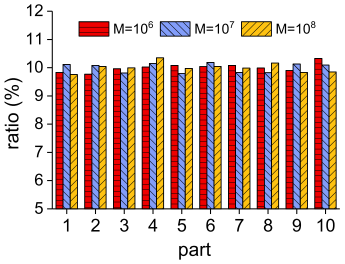


Figure 22: Data distribution of remaining elements in final array

we can increase the attendance of traffic police at these places. Figure 20 shows the corresponding actual cardinalities and latencies of the query as we vary the expected cardinalities of the non-equi-join operator. From the result, we can see that the actual cardinalities are exactly the same as the expected ones. The query latency increases linearly with the size of expected cardinalities due to the groupby and orderby operators involve more intermediate tuples output by the non-equi-join operator. Overall, Touchstone can well support the non-equi-join workload with the complicated predicates.

5.5 Insight into FK Constraint Adjustments

In Algorithm 3, we adjust the FK constraints by adding rules to avoid the appearance of mismatch case. About the adjustment, Figure 21 presents the maximum number of entries in the join information table and the maximum number of rules added in the FK constraints during the table generation as we vary the number of queries with $SF = 1$ and $SF = 1000$. The input queries are loaded in order of their serial numbers. For $SF = 1$, there are two groups of experimental results. In Figure 21, we can see that 1) for specific number of input queries and scale factor, the maximum numbers of entries and rules are both random, which are depended on the order of constraint chains and the result of query instantiation; 2) the maximum numbers of entries and rules increase obviously until reaching 10 input queries. Because for these two numbers, the target (or related) foreign key is within table *Order* of TPC-H, and there is no cardinality constraint over the equi-join operator associated with this foreign key in Q_{11} to Q_{16}; and 3) both two maximum numbers have a leap when the number of input queries changed from 6 to 10, and it is consistent with the result shown in Figure 14. Overall, the maximum numbers of entries and rules are both small, i.e., 40 and 41 respectively, for 16 queries of TPC-H workload.

5.6 Effectiveness of Compression Method

In subsection "Management of Join Information" of Section 4, we have presented the compression method for efficiently maintaining join information table. We verify the randomness of the values managed by the join information table in this part. The problem can be abstracted as follows. Supposing there are N elements which arrive one by one, and the size of N is not known in advance. We can only store L elements considering the storage cost, and these elements are required to be uniformly distributed among the N elements as much as possible. In general, L is much less than N. We simulate our compression method for above scenario by experiment. The size of L is set to 10^4, and the size of N varies from 10^6 to 10^8. The first L arrived elements are stored in an array. For i-th ($i > L$) element, it has the probability L/i to replace an element at any random position in the array. When all N elements are processed, we count the data distribution of the remaining L elements in the array, and the result is shown in Figure 22. We divide the original N elements into 10 parts evenly. For each part, we calculate the ratio of the number of its elements within final array to number L. From the result, we can see that the ratio for each part is approximately equal to 10%, so the remaining L elements are uniformly distributed in the original N elements. Our method is good for ensuring the randomness of generated data.

5.7 Performance Deviation of Schema-Aware Data Generation

Figure 23 shows the performance deviations of all TPC-H queries for a randomly generated database instance. The random database instance is generated by a schema-aware data generator with the inputs of database schema, data characteristics for all non-key columns and reference distributions between foreign keys and primary keys. We randomly select three groups of query parameters, and obtain the corresponding latencies for all queries respectively. From the result, we can see that the

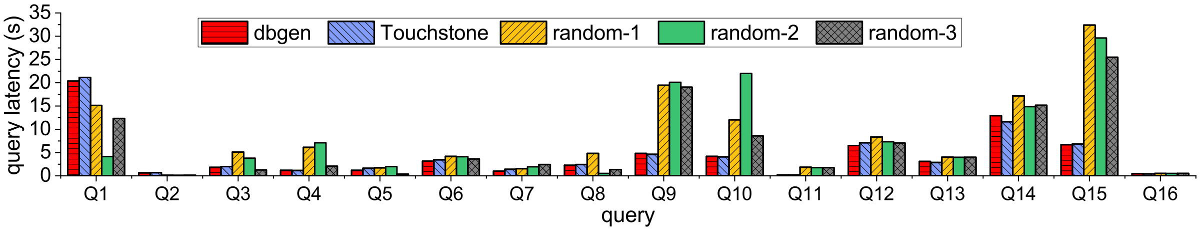


Figure 23: Performance deviation of schema-aware data generation

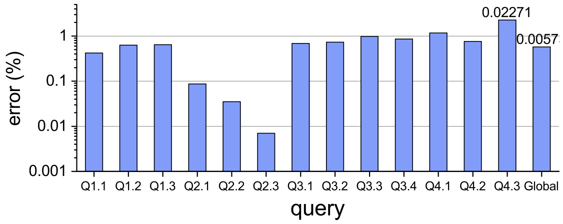


Figure 24: Relative error for each query of SSB

performance deviation of the vast majority of queries is obvious for the randomly generated database instance, which strongly validates the necessity of query-aware data generation.

5.8 Evaluation over Workloads of Star Schema Benchmark

The Star Schema Benchmark (SSB) [19] is a benchmark designed to measure query performance for analytical applications, where the tables are organized with a star schema. The workload of SSB contains 4 groups of queries, i.e., Q_1 to Q_4, with 13 queries, i.e., $Q_{1.1-1.3}$, $Q_{2.1-2.3}$, $Q_{3.1-3.4}$ and $Q_{4.1-4.3}$, which are all included in our experiments.

Figure 24 presents the relative errors for all 13 queries of SSB with $SF = 1$. We can see that the maximum error among 13 queries is 2.3%, and the global relative error of all queries is 0.57%. Figure 25 shows the total running time of *Touchstone* as we vary the number of input queries with $SF = 500$. *Touchstone* is deployed on 8 nodes. As shown in Figure 25, the total runtime grows slowly as we input more queries. Figure 26 presents the total running time of *Touchstone* under different scale factors over 8 nodes. Figure 27 shows the data generation throughputs per node of *Touchstone* as we vary the number of nodes with $SF = 500$. The input workloads of Figure 26 and Figure 27 both include all 13 SSB queries. From the results, it can be seen that *Touchstone* is linearly scalable to data size and multiple nodes. Overall, *Touchstone* can support the workload of Star Schema Benchmark very well.

6 Related Work

There are many data generators [7, 12, 11, 4, 21, 24, 1, 9] which only consider the data characteristics of the target database. For example, Alexander et al. [4] proposes pseudo-random number generators to realize the parallel data generation. Torlak [24] supports the scalable generation of test data from a rich class of multidimensional models. RSGen [21] can reverse the database metadata, especially the statistics, to generate a synthetic DBI with the similar data characteristics. However, all these data generators can not generate test databases with the specified workload characteristics on target queries.

There are query-aware data generators [6, 14, 5, 15], among which [6, 14, 15] are a series of work. QAGen [6] is the first query-aware data generator, but for each query it generates an individual DBI and its CSP (constraint satisfaction program) has the usability limitations as declared in experimental results. WAGen [14] makes a great improvement that it generates m ($\leq n$) DBIs with n input queries, but WAGen can't guarantee that only one DBI is generated and still has CSP performance problem. Though MyBenchmark [15] has done a lot of performance optimization, generating one DBI can not be promised for multiple queries and the performance is still unacceptable for the generation of terabyte scale database. Moreover, for integrating multiple symbolic databases generated by QAGen corresponding to input queries, huge constrained bipartite graphs (CBG) are constructed by [14, 15] which consume a lot of memory, and the integrating algorithm based on the CBG is difficult to parallelize. DCGen [5] uses a novel method to represent data distribution with ideas from the probabilistic graphical model. But DCGen is weak in support of foreign key constraint, and it cannot easily support parallel data generation in a distributed environment. Additionally, it may be hard to use the techniques introduced in DCGen for the great manual work of constructing input cardinality constraints, if you don't have a DBI meets the test requirements and its assumption, e.g., the

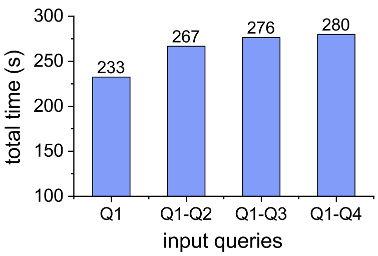


Figure 25: Total running time vs. input queries (SSB)

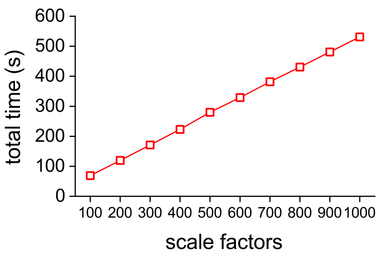


Figure 26: Scalability to data scale (SSB)

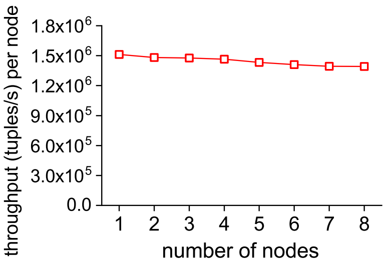


Figure 27: Scalability to multiple nodes (SSB)

cardinality of the column is small, such as dozens.

There are some interesting non-relational data generators [18, 8, 13, 20, 10]. For example, Olston et al. [18] introduces how to generate example data for dataflow programs. Sara [8] generates structural XML documents. [13, 20] are synthetic graph generators. Chronos [10] can generate stream data for real time applications. In addition, there are query generation works [17, 16] which are partly similar to us, but they generate queries satisfying the specified cardinality constraints over an existing DBI. Moreover, the dataset scaling works [23, 26] can serve part of our targets, which scale up/down a given DBI with similar column correlations.

7 Discussion and Conclusion

limitations. *Touchstone* aims to support the most common workloads in real world applications. Below we list the scenarios that we cannot support currently. (1) *Touchstone* does not support filters on key columns. Primary and foreign keys are identifiers of tuples and generally have no physical meaning, so the filters which are representations of business logics usually do not involve key columns. (2) Equality constraints over filters involving multiple columns are not supported in *Touchstone*. The equality predicate with multiple columns for filter is a very strict constraint, and has not been found in workloads of standard benchmarks. (3) Equi-joins on columns with no reference constraint are not supported in our work. This is because the equi-join is usually applied on the pair of primary and foreign keys in practical workloads, which is also the assumption of many works [5, 25, 26]. (4) *Touchstone* does not support the database schema with cyclic reference relationship. In our data generation process, generating foreign keys requires the join information tables of corresponding referenced primary keys, so the primary-foreign-key dependencies must form a direct acyclic graph (DAG), which is also the precondition of DCGen [5].

Privacy issue. Our work can help to protect privacy to some extend by removing query parameter values or using approximate query intermediate cardinalities. However, if the database statistics and workload characteristics are strictly related to privacy issues in some cases, it will not be a good way to use this kind of workload-aware data generators for performance testing.

In this paper we introduce *Touchstone*, a query-aware data generator with characteristics of completely parallelizable and bounded usage to memory. And *Touchstone* is linearly scalable to computing resource and data scale. Our future work is to support more operators, e.g., intersect and having, for covering the complex queries of TPC-DS, which has not be well supported by any existing query-aware data generation work. In order to make our work repeatable and referenceable, we make the source code and running examples of this paper available in github [2].

Acknowledgements: Rong Zhang and Aoying Zhou are supported by National Key Research and Development Plan Project (No. 2018YFB1003303), and National Science Foundation of China (NSFC) (No.61332006, 61672233 and 61732014). And this work was done when Zhenjie Zhang and Xiaoyan Yang were with Advanced Digital Sciences Center, supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

References

[1] DTM data generator. http://www.sqledit.com/dg/.

[2] Technical report, running examples and source code of Touchstone. https://github.com/daseECNU/Touchstone.

[3] TPC-H benchmark. http://www.tpc.org/tpch/.

[4] Alexandrov, A., Tzoumas, K., and Markl, V. Myriad: scalable and expressive data generation. *Proceedings of the VLDB Endowment 5*, 12 (2012), 1890–1893.

[5] Arasu, A., Kaushik, R., and Li, J. Data generation using declarative constraints. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (2011), ACM, pp. 685–696.

[6] Binnig, C., Kossmann, D., Lo, E., and Özsu, M. T. Qagen: generating query-aware test databases. In *Proceedings of the*

2007 ACM SIGMOD international conference on Management of data (2007), ACM, pp. 341–352.

[7] Bruno, N., and Chaudhuri, S. Flexible database generators. In *Proceedings of the 31st international conference on Very large data bases* (2005), VLDB Endowment, pp. 1097–1107.

[8] Cohen, S. Generating xml structure using examples and constraints. *Proceedings of the VLDB Endowment 1*, 1 (2008), 490–501.

[9] Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P. J. Quickly generating billion-record synthetic databases. In *ACM SIGMOD Record* (1994), vol. 23, ACM, pp. 243–252.

[10] Gu, L., Zhou, M., Zhang, Z., Shan, M.-C., Zhou, A., and Winslett, M. Chronos: An elastic parallel framework for stream benchmark generation and simulation. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on* (2015), IEEE, pp. 101–112.

[11] Hoag, J. E., and Thompson, C. W. A parallel general-purpose synthetic data generator. *ACM SIGMOD Record 36*, 1 (2007), 19–24.

[12] Houkjær, K., Torp, K., and Wind, R. Simple and realistic data generation. In *Proceedings of the 32nd international conference on Very large data bases* (2006), VLDB Endowment, pp. 1243–1246.

[13] Leskovec, J., Chakrabarti, D., Kleinberg, J., and Realistic, C. F. Mathematically tractable graph generation and evolution, using kronecker multiplication european conf. on principles and practice of know. dis. *Databases (ECML/PKDD)* (2005).

[14] Lo, E., Cheng, N., and Hon, W.-K. Generating databases for query workloads. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 848–859.

[15] Lo, E., Cheng, N., Lin, W. W., Hon, W.-K., and Choi, B. Mybenchmark: generating databases for query workloads. *The VLDB Journal 23*, 6 (2014), 895–913.

[16] Mishra, C., and Koudas, N. Interactive query refinement. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (2009), ACM, pp. 862–873.

[17] Mishra, C., Koudas, N., and Zuzarte, C. Generating targeted queries for database testing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 499–510.

[18] Olston, C., Chopra, S., and Srivastava, U. Generating example data for dataflow programs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data* (2009), ACM, pp. 245–256.

[19] O'Neil, P., O'Neil, E. J., and Chen, X. The star schema benchmark (ssb), 2009. revision 3.

[20] Pham, M.-D., Boncz, P., and Erling, O. S3g2: A scalable structure-correlated social graph generator. In *Technology Conference on Performance Evaluation and Benchmarking* (2012), Springer, pp. 156–172.

[21] Shen, E., and Antova, L. Reversing statistics for scalable test databases generation. In *Proceedings of the Sixth International Workshop on Testing Database Systems* (2013), ACM, p. 7.

[22] Syrjänen, T. Logic programs and cardinality constraints–theory and practice.

[23] Tay, Y., Dai, B. T., Wang, D. T., Sun, E. Y., Lin, Y., and Lin, Y. Upsizer: Synthetically scaling an empirical relational database. *Information Systems 38*, 8 (2013), 1168–1183.

[24] Torlak, E. Scalable test data generation from multidimensional models. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (2012), ACM, p. 36.

[25] Zamanian, E., Binnig, C., and Salama, A. Locality-aware partitioning in parallel database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 17–30.

[26] Zhang, J., and Tay, Y. Dscaler: Synthetically scaling a given relational database. *Proceedings of the VLDB Endowment 9*, 14 (2016), 1671–1682.

A Calculation of ratio and error

We give the following sample constraint chains of a table for illustrating the ratio adjustment in Algorithm 3.
①: FK[fk, rpk, 20%]
②: FILTER[$c \leq 6$] → FK[fk, rpk, 10%]
③: FILTER[$c > 3$] → FK[fk, rpk, 30%]

There are three constraint chains of the table, where the target table has one foreign key fk, and the referenced primary key for fk is rpk. The column c of the table is uniformly distributed in range (0,10], so there are 60%, 70% and 30% of the tuples satisfying the predicates $c \leq 6$, $c > 3$ and both of them, respectively. The cumulative probabilities of statuses T and F for a FK constraint can be calculated along the located constraint chain. For the second FK constraint, its cumulative probabilities of statuses T and F are $60\% \times 10\% = 6\%$ and $60\% \times 90\% = 54\%$, respectively. Suppose there are two rules in the three FK constraints, and they are rule $[FT \leftarrow T]$ in the second FK constraint and rule $[TFT \leftarrow FT]$ in the third FK constraint. All conflicting rules in the FK constraints are removed, and the details are showed in Appendix.B.

Algorithm 4 Ratio adjustment

```
Input: FK constraint $C_i$ and the added rules $R$
 1: $p_T \leftarrow 0$, $p_F \leftarrow 0$
 2: for all rule $r \in R$ do
 3:     $\eta \leftarrow$ condition of $r$, $p_\eta \leftarrow 1$
 4:     for all status $s \in \eta$ do
 5:         $p_\eta \leftarrow p_\eta \times p_s$
 6:     end for
 7:     if target status of $r$ is $T$ then
 8:         $p_T \leftarrow p_T + p_\eta$
 9:     else
10:         $p_F \leftarrow p_F + p_\eta$
11:     end if
12: end for
13: $p'_T \leftarrow C_i.p_T - p_T$, $p'_F \leftarrow C_i.p_F - p_F$
14: $C_i.ratio \leftarrow p'_T/(p'_T + p'_F)$
```

In Algorithm 4, we present the detailed calculation process of ratio adjustment. Firstly, the algorithm initializes (line 1) two counters, i.e., p_T and p_F, for counting the occupied cumulative probabilities of the statuses T and F which are forcibly generated by the rules. Then, the algorithm iterates over all rules within FK constraint (line 2). For each rule, the algorithm calculates (lines 3-6) the occurrence probability p_η of its condition η (e.g., for rule $[TFT \leftarrow \boldsymbol{FT}]$, η is $\boldsymbol{FT}$). And p_η is the occupied cumulative probability of the target status indicated by the leftmost bit of the rule (e.g., for rule $[\boldsymbol{T}FT \leftarrow FT]$, the target status is $\boldsymbol{T}$). For calculating p_η, the algorithm multiplies (line 5) the probabilities, i.e., p_s, of all statuses in η. After obtaining p_η for the condition of the rule, the algorithm updates the corresponding counter according to the target status of the rule (lines 7-10). Next, when all rules have been iterated, the algorithm calculates (line 13) the remaining cumulative probabilities p'_T and p'_F for statuses T and F of current FK constraint C_i, where $C_i.p_T$ and $C_i.p_F$ are the original cumulative probabilities of the statuses T and F respectively. Finally, the ratio of current FK constraint $C_i.ratio$ is adjusted according to the remaining cumulative probabilities.

The probability of each status p_s is determined according to the following two considerations. Considering the rules added in previous FK constraints, the algorithm traverses all statuses in η. For any status in η, if it is forcibly generated by one rule applied on the previous statuses in η, the p_s is set as 1, otherwise p_s is set as the remaining cumulative probability of the corresponding FK constraint. Considering the involved FILTER constraints, the algorithm revises the probability p_s based on the full satisfaction of all related filters.

We present a running example of ratio adjustment using the sample constraint chains. 1) For the first FK constraint, because there is no rule in it, its remaining cumulative probabilities p'_T and p'_F are 20% and 80% respectively, and its ratio is still 20%. 2) For the second FK constraint, there is a rule $[FT \leftarrow T]$ in the constraint, and the rule's condition $\eta = T$. The p_s of status T in the η is set as $20\% \times 60\% = 12\%$, where 20% is the corresponding remaining cumulative probability p'_T of the first FK constraint, 60% is the ratio of tuples satisfying the predicate $c \leq 6$, and 12% is the revised cumulative probability under the satisfaction of FILTER[$c \leq 6$]. Since the target status of the rule $[FT \leftarrow T]$ is F, the counter p_F is updated to 12%. Then, the remaining cumulative probabilities $p'_T = 6\%$ and $p'_F = 54\% - 12\% = 42\%$, and the ratio is adjusted to $6\%/(6\% + 42\%)$. 3) For the third FK constraint, there is a rule $[TFT \leftarrow FT]$ in the constraint, and the rule's condition $\eta = FT$. For $\eta = FT$, the p_s of status F in the η is set as 1 due to the existence of the rule $[FT \leftarrow T]$ in the second FK constraint, and the p_s of status T in the η is set as $20\% \times 30\% = 6\%$, where 20% is the corresponding remaining cumulative probability p'_T of the first FK constraint, 30% is the ratio of tuples satisfying the predicates $c \leq 6$ and $c > 3$, and 6% is the revised cumulative probability under the full satisfaction of FILTER[$c \leq 6$] and FILTER[$c > 3$]. So final $p_\eta = 1 \times 6\% = 6\%$. Accordingly, the remaining cumulative probabilities $p'_T = 70\% \times 30\% - 6\% = 15\%$ and $p'_F = 70\% \times 70\% = 49\%$, and the ratio is adjusted to $15\%/(15\% + 49\%)$.

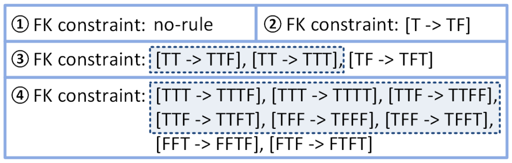

① FK constraint: no-rule
② FK constraint: [T -> TF]
③ FK constraint: [TT -> TTF], [TT -> TTT], [TF -> TFT]
④ FK constraint: [TTT -> TTTF], [TTT -> TTTT], [TTF -> TTFF], [TTF -> TTFT], [TFF -> TFFF], [TFF -> TFFT], [FFT -> FFTF], [FTF -> FTFT]

Figure 28: A sample adjustment of Algorithm 3 for illustrating the generation of conflicting rules

For a FK constraint C_i, if the adjusted ratio is not between 0 and 1, namely one of the p'_T and p'_F is less than 0, the adjustment for constraint C_i is illegal and the introduced error is defined as $e_i = |Min\{p'_T, p'_F, 0\}| \times s^T$, where s^T is the size of the involved table. Therefore, the error caused by all illegal adjustments is $e = \sum_{1 \leq i \leq k} e_i$, where k is the number of FK constraints.

B Discussion of conflicting rules

As described in Algorithm 3, we adjust the FK constraints by adding rules and accordingly revising the ratio for eliminating the mismatch cases. During the adjustment, there may be some conflicting rules added in the FK constraints. For two rules, if they have the same conditions and the opposite target statuses, we call them conflicting rules. Figure 28 presents the generation process of conflicting rules. For the first FK constraint, there is no rule added in it. For the second FK constraint, there is a rule $[FT \leftarrow T]$ built for it. For the third FK constraint, there are three rules, in which two rules $[FTT \leftarrow TT]$ and $[TTT \leftarrow TT]$ are conflicting rules. For the fourth FK constraint, there are eight rules, in which six rules walled by the dotted line are conflicting rules. We can see that if there is a rule built for i-th FK constraint, correspondingly there will be two conflicting rules built for $(i+1)$-th FK constraint. This is because the rule building for each FK constraint in Algorithm 3 has not considered the added rules in previous FK constraints. The conflicting rules added in FK constraints would never be used as declared in Proposition.1.

Proposition 1 *In the tuple generation, it would not have two conflicting rules applied at the same time for any FK constraint, and all conflicting rules in the FK constraints would never be applied.*

Proof 1 *For the first FK constraint, if there are two conflicting rules $[F \leftarrow Null]$ and $[T \leftarrow Null]$, it indicates* *that there is no referenced primary key with the status T or F, while the status of primary key can only be T or F. Therefore, conflicting rules would not exist in the first FK constraint unless the input of Touchstone is illegal. For i-th FK constraint (i ≥ 2), if there are two conflicting rules [$Fs_{i-1}...s_2s_1 \leftarrow s_{i-1}...s_2s_1$] and [$Ts_{i-1}...s_2s_1 \leftarrow s_{i-1}...s_2s_1$], the joinability statuses $Fs_{i-1}...s_2s_1$ and $Ts_{i-1}...s_2s_1$ must be both missed in the join information table. So the joinability status prefix $s_{i-1}...s_2s_1$, too, must be missed for the $(i-1)$-th FK constraint. Then it shall have a rule [$\bar{s}_{i-1}s_{i-2}...s_2s_1 \leftarrow s_{i-2}...s_2s_1$] (or [$\bar{s}_{i-1} \leftarrow Null$] when $i = 2$) added in the $(i-1)$-th FK constraint, where $\bar{s}_{i-1} = F$ for $s_{i-1} = T$, vice versa. Therefore, it would not have two conflicting rules applied at the same time or even any one of the conflicting rules applied for the i-th FK constraint, because there must be a rule applied in the $(i-1)$-th FK constraint preventing it happening.*

For example, in Figure 28, the two conflicting rules $[FTT \leftarrow TT]$ and $[TTT \leftarrow TT]$ in the third FK constraint would never be applied, because the condition TT can not be met due to the existence of the rule $[FT \leftarrow T]$ in the second FK constraint.