A synthetic query-aware database generator

Maria Jose Cepeda Garcia
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
mc4276@cs.rit.edu

Abstract—In database applications and DBMS products, query performance is mandatory. Synthetic databases are a common approach for testing query performance. However, the generation of meaningful synthetic data for testing is intractable in general. We present a framework for the automatic generation of synthetic databases for a set of user requirements defined on test queries. Our solution is evaluated on TPC workloads and the results show its feasibility, generating a set of databases which comply with the user requirements.

Index Terms—synthetic data; query-aware database; database generator; test database; DBMS testing; symbolic query processing

I. INTRODUCTION

Synthetic databases are a common approach for testing query performance. We study the problem of generating synthetic databases that comply with some user requirements such as data characteristics (e.g. values distribution of an attribute) and test SQL queries. Applications of synthetic databases include:

1) DBMS Testing: The addition or modification of components in database engine systems (DBMS) requires to verify the correctness and performance of the system using a wide range of test cases in different scenarios. As such, it may be necessary the generation of synthetic databases with different characteristics [2, 3, 10]. Moreover, to achieve accurate results over the performance of those components, it is necessary to control the workload in every query operator. This is possible through cardinality constraints use [3]. For example, to test a new memory manager and its performance with multiple hash join operators, we may require a database that enforces a particular intermediate cardinality result on those operators for a given test query [1].

2) Database application testing: Some organizations outsource the development of their applications. As a consequence, developers may not have access to the real database to test their SQL queries. These situations require the generation of synthetic databases to mimic the real database. Furthermore, synthetic database generators can be used to create an upscaled version of the original database to perform stress testing on the application.

3) Benchmarking: Standards benchmarks like TPC are common for deciding between competing DBMS products. They are also widely used for the comparative evaluation of new research works against their state-of-the-art [1, 2, 10, 11]. However, there may be cases where those standards benchmarks do not capture some scenarios of interest for a tester. That motivates the generation of synthetic databases.

We analyze the problem of generating synthetic databases through the study of MyBenchmark [10]. MyBenchmark is a synthetic workload-aware database generator. It generates a minimal set of databases that together satisfy a set of user queries. In addition, this generator also allows the specification of the size of the tables along with basic data characteristics (e.g correlations, customers under a certain age are not allowed to buy specific products). MyBenchmark is an extension of QAGen [2]. QAGen and other previous works, also consider user queries [9, 11]. Although, they generate a new independent database for every query (single query-aware). Applications of these works are more limited. For example, they are not suitable for database application testing since users would be forced to switch between $n$ databases to test $n$ SQL queries. This paper presents our implementation of MyBenchmark. Furthermore, since MyBenchmark uses QAGen as a core, we also present our implementation of QAGen.

The rest of the paper is organized as follows: Section 2 discusses related work. In Section 3, we present our system’s architecture. Section 4 describes our implementation for MyBenchmark and QAGen’s algorithms. The experimental evaluation is shown in Section 5. Finally, conclusions and future work are discussed in Section 6.

II. RELATED WORK

In DBMS testing, we verify the correctness and performance of the components of database engines. To test those components, it may be necessary to generate a database with specific requisites. For example, the testing of a new outer join implementation may need a database with correlation between tables (foreign keys), and a concrete value distribution of the data on an attribute (e.g. Zipf’s distribution). To test performance, the size of the database tables is also important. Finally, it is also required that the database complies with our test queries. Existing synthetic database generators allow the specification of the size and the data characteristics of the tables (value distribution, inter/intra table correlation) [3, 6, 7, 14]. Unfortunately, they do not take into account user queries [1, 2]. As
a result, some of these user queries may retrieve no (or not enough) results when executed over a synthetic database.

Manilla et al. [11] present an approach to generate synthetic databases taking user queries into account. Their solution generates small datasets for simple user queries. Bruno et al. [4] generate databases that comply with cardinality constraints. A cardinality constraint specifies the cardinality (size) of the output of a query. Using cardinality constraints, we can specify the size and the data characteristics of the tables as well as user queries. For example, the distribution of values in an attribute can be represented by a histogram, and each bucket in the histogram can be represented as a set of cardinality constraints. Bruno et al. focus on generate values for parametric queries. These generated values fulfill a set of cardinality constraints. However, our goal is the generation of databases rather than query parameters.

Bruno et al. motivated further research considering cardinality constraints [1, 2, 9, 12]. QAGen [2] uses cardinality constraints to generate a synthetic database that satisfies an user query. Given a schema definition $D$, a query $Q$, and a set of cardinality constraints $C$ over $Q$, QAGen generates a database that fulfills $C$. In a first phase, QAGen creates a symbolic database based on the schema definition. In this symbolic database, the data consist of placeholders. Then, QAGen processes the cardinality constraints using a constraint solver to create concrete data that satisfies the input query. Those data are then populated into the database. QAGen as well as other research [9, 11] overcome this limitation by generating a single synthetic database that satisfies several user queries at the same time. Their algorithms are probabilistically approximate. Therefore, the resulting database may not satisfy all the constraints provided by the user. MyBenchmark [10] also addresses the limitation of generating a new independent database for each user query. This tool uses QAGen [2] as a core. In a first step, $n$ independent synthetic databases are created for $n$ user queries using QAGen. Then, an integration module matches the data contained in such databases to heuristically identify $m \leq n$ databases that together satisfy all the constraints.

III. Overview

The definition of the problem we address in this paper is as follows: given a database schema definition $D$ and a set of queries $Q = \{Q_1, Q_2, ..., Q_n\}$, where every query $Q_i$ is annotated with a set of constraints $C_i$ (e.g. value distribution, cardinality constraint). Our solution generates $m \leq n$ databases $DB_1, DB_2, ..., DB_m$ such that: 1) all databases $DB_i$ $(1 \leq i \leq m)$ conform $D$, and 2) the resulting cardinalities $C'_i$ of posing $Q_i$ over one of the databases $DB_j$ approximate satisfy $C_i$ [10]. These approximate cardinalities are sufficient for most applications of database generators [1, 4]. Assume a tester annotates a select query with a cardinality constraint of five tuples over a database of 1 GB of data. In this case, it is acceptable to generate a database where such query returns eight tuples.

The data generation process consists of three phases; 1) symbolic query processing (SQP) phase, 2) symbolic database integration (SI) phase, and 3) data instantiation (DI) phase. The SQP phase executes the set of input queries $Q = \{Q_1, Q_2, ..., Q_n\}$ and generates a set of symbolic databases $SDB = \{SDB_1, SDB_2, ..., SDB_n\}$, such that $Q_i(SDB_i)$ approximately meets the constraints $C_i$. The SQP engine executes the queries applying the concept of symbolic processing [8]. Symbolic processing is a technique where program variables values are symbols or placeholders instead of concrete values. SQP phase incorporates this technique into the traditional query processing. As a consequence, after executing the queries, the resulting databases contain symbols rather than concrete data (see Figure 5c). These databases are called symbolic databases (SDB). The SI phase then, try to integrate those SDBs. The aim of SI phase is to reduce the numbers of SDBs that together satisfy all the requisites from the set of input queries.

The data instantiation phase is the final phase of the data generation process. Its goal is to generate concrete values for all the symbols stored in the SDBs. The data instantiator reads the data from the SDBs created by the SI phase and generates concrete data invoking a constraint solver [13]. The instantiated data is then inserted into the target database.

Figure 1 shows the architecture of our system [10]. It is compound by three main modules; 1) symbolic query engine, 2) symbolic database integrator, and 3) data instantiator. Each of the modules are related to a different phase in the data generation process respectively.

A. Symbolic query processor

Input queries are annotated queries in relational algebra expressions. Users can define constraints over every operator on the query as well as on the base tables. The definition of constraints on every operator is compulsory, users may define constraints for just some operators. In that case, the symbolic query engine will process those operations with default values. SQP engine takes leverage of the traditional query processing executing the queries in a bottom-up manner. That creates a pipeline of relational algebra operations where every operator is implemented as an iterator, and the data flows from the base tables up to the root of the query plan [5]. Although, there are cases where an operator must perform like a block operator (see case two for equi-join operator IV-A2b).
Every relational algebra operation processes the input data from its child operator (one tuple at a time) applying its own semantic and imposes the constraints on that operator into the SDB. Furthermore, every operator controls its cardinality constraint, so the parent operator has the correct data. Let’s consider query Q3 from Figure 4a. The selection operation has a cardinality constraint of two tuples. The first time, the selection operation reads tuple t1 from Customer relation, associates a positive constraint to the symbol c_age1 (c_age1 ≤ 20), and returns t1 to its parent. The same process is repeated for tuple t2. When the operator is invoked the third time, its reads tuple t3 and associates a negative constraint to the symbol c_age3 (c_age3 > 20). Then it returns null to its parent since the cardinality constraint (two tuples) has been reached.

In this version, our solution supports selection (σ), projection (π), and equi-join (▷◁) operations. Nevertheless, this set of operators allows the definition of fairly complex queries [2]. All operations implement three methods: open(), getNext(), and close(). Section IV-A explains in details the algorithm of getNext() method for every operator. open() and close() are omitted because of their simplicity.

B. Symbolic database integrator

The problem of generating one single symbolic database that complies with the set of input queries is NP-Hard [10]. Assume the symbolic execution of queries Q1 and Q2 from Figure 2 over the same SDB. These two queries are simple selection queries over the same attribute a from table R (Figure 2c). The cardinality constraint for the selection operator is one tuple for both queries. The symbolic execution of Q1 associates a positive selection predicate over tuple t1 (a1 ≤ 20), and a negative selection predicate over tuple t2 (a1 > 20). When Q2 is executed, tuple t1 has associated the positive selection predicate (a1 > 30), and tuple t2 has associated the negative selection predicate (a1 ≤ 30). In this case, tuple t1 has associated the predicates (a1 ≤ 20 and a1 > 30). These two predicates yield to a contradiction because together they are unsatisfiable (a1 ≤ 20 and a1 > 30).

IV. IMPLEMENTATION

This section describes the algorithms implemented for every module in our system.
A. Symbolic query processing

For our symbolic query processing module, we have followed the algorithms described by QAGen [2]. However, we implement some relational algebra operations slightly different. In this section, we present our algorithms for the selection and join operations. To implement these operations, we have considered whether the input data is pre-grouped or not. An input data is not pre-grouped in an attribute $a_i$ if and only if all symbols for that attribute $a_i$ are distinct. Pre-grouped data increase the hardness of the symbolic execution of an operator. For example, the complexity of an equi-join operator is NP-Hard when the data is pre-grouped on the join attributes (see case two for equi-join operator IV-A2b). Figure 4 and 5b are examples of no pre-grouped input data. Figure 5c shows an Order table pre-grouped on the o_cid attribute.

For this version, our symbolic query processing engine only considers left-most query plans.

1) Selection operator: For this paper, since every logical formula can be converted into an equivalent formula in conjunctive normal form (CNF). We assume the selection predicate is always in CNF.

a) Case 1 (pre-grouped): This case happens when the child operator is a join, and the data is pre-grouped on an attribute in the selection predicate. Most of the time, this case can be avoided moving the selection operation in the query plan just right before the join operation (this change in the tree query plan does not change the final result of the query). The symbolic execution of the selection is NP-Hard in this scenario [2]. Due to its complexity and the fact that it is a rare case, we do not support it.

b) Case 2 (no pre-grouped):

Input: Cardinality constraint $c$ (default value = child’s cardinality constraint)

Figure 4a is an example of this case. The selection operator has a table operator as a child. According to the symbolic execution of the Table operator, the data from the base table are unique. Therefore, the selection operator deals with distinct symbols for the attributes in the selection predicate. The symbolic execution of this case is as follows:

- The getNext() method gets the next tuple $t$ invoking the getNext() method from its child operator. If the cardinality constraint $c$ has not been reached, associate a positive selection predicate to tuple $t$. If $c$ has been reached, associate a negative selection predicate to tuple $t$.
- Positive selection predicate: For each symbol $s$ of tuple $t$ that participates in the selection predicate, insert a tuple $<s, t>$ in PTable. Return $t$ to its parent.
- Negative selection predicate: Once the cardinality constraint $c$ has been met, we need to enforce that the rest of tuples from the child operator do not satisfy the selection predicate. For such, fetch the remaining tuples $T^-\setminus T^-$ from its child operator. For each symbol $s$ of tuple $t \in T^-\setminus T^-$ that participates in the selection predicate, insert a tuple $<s, \neg p>$ in PTable. Return null to its parent.

Assume query Q3 from Figure 4 as the running example. In this query, the cardinality constraint of the selection operator is two tuples. The first two times the getNext() method is invoked, the cardinality constraint $c$ has not been reached. Tuples $t_1$ and $t_2$ have associated a positive selection predicate ($c\_age1 \leq 20$, $c\_age2 \leq 20$) and they are returned to the projection operator. The third time the method getNext() is invoked, $c$ has been reached. Tuple $t_3$ is associated with a negative selection predicate ($c\_age3 > 20$). This negative predicate prevents the Data Instantiator to generate a value for $c\_age3$ that satisfy the selection predicate. Figure 4c shows the content of PTable after processing Q3.

To improve the performance of this operator, instead of inserting a tuple for each getNext() method call. We store all those insert statements into a list. Once the cardinality has been reached, those insert statements along with the insert statements for the remaining tuples are sent to the database all together. Doing that, we are able to reduce the number of access to the database, and therefore, to speed up the symbolic execution process. We discuss in detail the benefits of this approach in Section V.

2) Equi-join operator: This operator has two children operators R and S, and the equi-join predicate $j \equiv k$ where $j$ is the primary key from R, and $k$ is the foreign key from S. Based on whether the data is pre-grouped w.r.t $k$ or not, we consider two different cases for the symbolic execution of this operator.

a) Case 1 (no pre-grouped w.r.t $k$):

Input: Cardinality constraint $c$ (default value = left child’s cardinality constraint), value distribution $d$ (options = [Uniform, Zipf], default value = Uniform)

Query Q4 from Figure 5a is an example of this case, where the data is not pre-grouped on attribute $k$. In query Q4, $k$ is the attribute o_cid from Order table (Figure 5b). For this case, users can specify the data value distribution $d$. This distribution defines how many tuples from S (frequency) are joined with a tuple from R. For example, Uniform distribution (default value) joins each tuple from R with roughly the same number of tuples from S. The symbolic execution of this case is as follows:

- open() method: Initialize the distribution generator, where the frequency is the cardinality constraint $c$, the number of samples $n$ is the cardinality constraint from R, and the distribution type is $d$. We use Apache Commons Math library to implement this generator. The generator creates
an array $A$. The values in $A$ represent the number of
tuples from $S$ that join with a tuple from $R$. For the $i$th call
on the getNext() method ($0 < i \leq n$), get the frequency
value $f_i$ from the $i$th position of $A$.

- **getNext() method:** If the cardinality constraint $c$ has not
  been reached:
  - Verify whether $f_i$ is zero or it has not been initialized. If so, get $f_i$ value from A.
  - Read in a tuple $r^+$ from $R$ invoking its getNext() method.
  - Read in a tuple $s^+$ from $S$ invoking its getNext() method. Decrease $f_i$ by one.
  - Perform a positive joining between $r^+$ and $s^+$. Return the joined tuple to its parent.

If the cardinality constraint $c$ has been reached, perform a negative joining and return null to its parent.

- **Positive joining:** For tuple $s^+$, replace the symbol $s^+, k$ for $r^+, j$. Now, attributes $k$ and $j$ shares the same symbol, enforcing the foreign key relationship between $R$ and $S$. This update must be propagated to the base table as well (executing an update SQL statement like “update base_table set $k = r^+, j$ where $k = s^+, k$”). Perform a join between tuples $r^+$ and $s^+$. Return the joined tuple to its parent.

- **Negative joining:** Once the cardinality constraint $c$ has been reached, we need to enforce the foreign key relationship for the remaining tuples $S^-$ from $S$. This update is done only in the database since this case returns null to its parent. For each tuple $s^- \in S^-$, randomly choose a symbol $j^-$ from the base table such that $j^- \notin R$, replace $s^-, k$ with $j^-$ (reuse update SQL statement from positive joining case).

Query 4 from Figure 5 has a join operator between the output of the selection operator over the Customer table and the base table Order. The cardinality constraint of the selection operator is two tuples and the cardinality constraint $c$ of the join operator is three tuples. The join operator has not specified the distribution value $d$, by default is Uniform distribution.

The first step is to create the distribution generator. For this example, the generator creates an array of length two (the cardinality constraint of $R$), with a total frequency of $3$ ($c$ value). Assume the generator creates the following array $A = [2, 1]$, meaning that the first tuple from the Customer table ($c_{id1}$) is joined with the first two tuples from the Order table ($o_{id1}, o_{id2}$), and the second customer ($c_{id2}$) is joined with the third tuple ($o_{id3}$). Once the cardinality constraint $c$ has been reached, the negative joining enforces the foreign key relationship for the tuple $o_{id4}$. This tuple must be joined with a customer not returned for the selection operator (in this example, customer $c_{id3}$). Figure 5b shows the content of the SDB right after the execution of both Table operators. Figure 5c shows the content of the SDB after the symbolic execution of the entire query Q4. Tuples in bold represent the output of the query.

\textbf{b) Case 2 (pre-grouped w.r.t $k$):}

\begin{itemize}
  \item \textbf{Input:} Cardinality constraint $c$ (default value = left child’s cardinality constraint)
  \end{itemize}

In this case, the input data from $S$ is pre-grouped on the attribute $k$. That happens as result of the value distribution of a preceding join operator. Figure 5c shows how Order table is pre-grouped on the attribute $o_{cid}$ after the symbolic execution of the join operator from Query Q4 (Figure 5a).

For this case, the problem of controlling the output cardinality constraint $c$ can be characterized as the Subset-sum problem [2]. The definition of this problem is as follows: given a set of integers $C = \{c_1, c_2, \ldots, c_n\}$ and an integer target $t$, find a subset $C^+ \subseteq C$ such that $\sum_{c_i \in C^+} c_i = t$. Let’s consider the data from the Order table (Figure 5c). The data is clustered in $n$ groups, where every group has $c_i$ tuples with the same value for $k$. The objective is to find a subset of clusters that sums the cardinality constraint $c$. The original Subset-sum problem tries to find a subset that exactly sums $t$. However, in our case, we have implemented a dynamic programming algorithm that approximate satisfies $c$. This algorithm is called Overweight Subset-sum problem.
and returns a subset with sum greater than or equal to the target \( t \) without exceeding an approximation ratio of \( \epsilon \) [9]. Assume the set \( C = \{1, 3, 5, 13, 44\} \) and the target \( t = 7 \). In this case, with an approximation ratio of \( \epsilon = 0.1 \), this algorithm returns the subset \( C^+ = \{3, 5\} \) that sums 8. This approximate cardinality is sufficient for most applications of database generators [1, 4]. The complexity of this dynamic programming algorithm is \( O(\frac{n}{\epsilon^2}) \) [9].

When the data is pre-grouped, the join behaves like a blocking operator. In order to control the output cardinality \( c \), it is necessary to read first the entire input data from \( S \). Assume we perform a join when \( R \) child has three tuples and \( S \) child has the data from Figure 6. The join attribute \( j \) is \( c_{id} \) and \( k \) is \( a_{cid} \). The data from \( S \) is pre-grouped in three clusters (\( c_{id1} = 5 \) times, \( c_{id2} = 3 \) times, \( c_{id3} = 1 \) time) and the cardinality constraint is \( c = 8 \). The symbolic execution of this case is as follows:

- open() method: (a) Read in all tuples from \( S \) (blocking operator), (b) compute the clusters in \( S \) (\( c_{id} = 5 \), \( c_{id} = 3 \), \( c_{id} = 1 \)), (c) invoke the approximate Subset-sum algorithm. The algorithm returns the subset \( C^+ = \{c_{id1}, c_{id2}\} \) that sums 8. If the approximate algorithm does not find a solution, return an error to the user.
- getNext() method: If cardinality constraint \( c \) has not been reached, perform positive joining. Otherwise, perform negative joining.
- Positive joining: For each symbol \( k_i \in C^+ \), read in all tuples \( S^+ \) from \( S \) with \( k = k_i \). Read in a tuple \( r^+ \) from \( R \) invoking its getNext() method, replace the symbol \( k \) from \( S^+ \) with the \( r^+, j \). Propagate this update in the database as well.
- Negative joining: Similar to the negative joining from case IV-A2a.

Notice that if the cardinality constraint \( c \) is equal to the input size of \( S \), then this case can be treated as case 1 (no pre-grouped w.r.t \( k \)) since all tuples from \( S \) must be joined with a tuple from \( R \). It is not necessary to invoke the dynamic programming algorithm.

To improve the performance of this operator, we follow the approach from the selection operation. On each getNext() method call, store the update statements from the positive joining in a list. Once the cardinality has been reached, send those statements along with the statements for the negative joining to the database.

B. Symbolic database integrator

After the SQP phase, \( n \) independent SDBs have been generated for the set of input queries \( Q = \{Q_1, Q_2, ..., Q_n\} \). The objective of this module is to reduce the number of SDBs that together satisfy the set of input queries.

The first task of this module is to define the order in which the \( n \) databases are integrated. For such, we create an integration plan tree (see Section III-B). Then, we traverse the tree in a bottom-up manner and try to integrate the SDBs in pairs. The integration is done table-by-table. MyBenchmark [10] models the problem of integrating two symbolic tables (with same schema definition and same size) as the problem of finding a satisfiable matching of size \( k \) on a constrained bipartite graph (CGB) \( G = (V_0, V_1, E) \) (see Section 3.2 from MyBenchmark).

We follow the same approach in our SI algorithm. Given two symbolic databases \( SDB_1 \) and \( SDB_2 \), we create a CBG \( G = (V_0, V_1, E) \) for every common table. Then, we look for a satisfiable matching of size \( |V_0| \). If that matching does not exist, \( SDB_1 \) and \( SDB_2 \) cannot be integrated. The number of vertices in \( V_0 \) is always equal to the number of vertices in \( V_1 \) (\( |V_0| = |V_1| \)). That is due to the fact that the number of tuples in both tables must be the same and the symbolic query processing always associates constraints for all tuples in a table. That is the reason because we look for a satisfiable matching of size \( |V_0| \). Every tuple from \( SDB_1 \) must be integrated with a tuple from \( SDB_2 \).

Figure 7 shows the symbolic integration between \( SDB_1 \) and \( SDB_2 \). Both SDBs consist of the same \( R \) table where every tuple has associated a predicate in the PTable. Notice that the content of PTable is different in each SDB. The goal is to integrate these SDBs without generating contradictory constraints. For example, to integrate tuple \( t_1 \) and \( t_5 \) is not possible since the conjunction of their predicates (\( a1 \leq 20 \&\& a1 > 30 \)) is unsatisfiable, and therefore, a contradiction. Figure 7c is the constrained bipartite graph that represents the integration between \( SDB_1 \) and \( SDB_2 \). Every vertex in the graph represents the set of predicates associated to a tuple. The edges represent the possible integration between two tuples.
For example, tuples t1 and t7 can be integrated into a single tuple since the formula \((a1 \leq 20 \land a1 \leq 30)\) is satisfiable.

For finding the \(k\) satisfiable matching, we have implemented a backtracking algorithm (Algorithms 1 and 2). This algorithm incrementally builds candidates to the satisfiable matching. Once a candidate solution reaches size \(k\), it checks whether it is indeed a satisfiable matching. If that candidate is not satisfiable, abandons that candidate (“backtracks”) and computes a new one. In our case, we just need to find one satisfiable matching of size \(k\), there is not necessary to compute all possible solutions. Therefore, if the algorithm finds a \(k\) satisfiable matching in the CBG, it can stop early. Figure 7d shows a satisfiable matching of size four for the integration between \(SDB_1\) and \(SDB_2\) (Figure 7a and 7b respectively). The complexity of our algorithm is \(O(n!)\) in the worst case scenario.

Algorithm 1 K_SAT_Match \((G, V_0, V_1, k)\)

▷ Input: (a) Constrained bipartite graph \(G\), (b) Left set of vertices from \(G\) \(V_0\), (c) Right set of vertices from \(G\) \(V_1\), (d) Size of the satisfiable matching \(k\)
▷ Output: Satisfiable matching of size \(k\)
1: sort \(V_0\) by the number of neighbors (asc order)
2: findMatching(G, \(V_0\), \(V_1\), \(k\), \(map\) := empty map, \(idx\) := 0, \(isSAT := false\));
3: if \((isSAT)\) then
4: \(\text{return} \map\)
5: \(\text{end if}\)
6: return null

C. Data instantiator

The data instantiated by this module must comply with: 1) the data type in the schema definition \(D\), 2) constraints in the schema definition \(D\) (e.g. not null, unique, check constraints), and 3) constraints from PTable. We use a constraint solver to generate data that satisfy all those requisites. The solver receives a set of predicates as input and returns concrete values for each symbol that participates in those predicates. The set of predicates is treated as a propositional formula. If that formula is unsatisfiable, the solver returns null. Nevertheless, this case is unlikely to happen since the objective of the symbolic integrator module is to integrate SDBs without contradictory constraints. Therefore, the set of predicates sent to the solver must be satisfiable.

For this module, we have implemented the algorithm proposed by QAGen [2]. QAGen uses a constraint solver written in C++ called Cogent. Unfortunately, this solver is no longer available. Our solution uses a constraint solver written in Java called Choco solver [13]. In the worst case scenario, the cost of a constraint solver call can be exponential to the size of the propositional formula. Although, we show in our experiments that this module scales linearly (see Section V).

Algorithm 2 findMatch \((G, V_0, V_1, k, map, idx, isSAT)\)

▷ Input: (a) Constrained bipartite graph \(G\), (b) Left set of vertices from \(G\) \(V_0\), (c) Right set of vertices from \(G\) \(V_1\), (d) Size of the satisfiable matching \(k\) (e) partial solution \(map\), (f) Pointer to \(V_0\) \(idx\), (g) Boolean variable \(isSAT\)
▷ Output: \(map\)
1: if \((\neg isSAT)\) then
2: if |\(map| = k\) then \(\triangleright \) possible matching found
3: \(isSAT := isSAT(\map)\);
4: \(\text{end if}\)
5: else
6: \(u := V_0[idx]\)
7: for \(v := \text{neighbors}(G, u)\) do
8: if \((v\) is free \(\land \neg isSAT)\) then \(\triangleright \) mark the vertex \(v\)
9: \(\text{map} \leftarrow (u, v)\)
10: \(isSAT := findMatch(G, V_0, V_1, k, map, idx + 1, isSAT)\)
11: if \((\neg isSAT)\) then
12: remove \((u, v)\) from \(map\)
13: \(\text{end if}\)
14: \(\text{end if}\)
15: \(\text{end for}\)
16: \(\text{end if}\)
17: \(\text{return} \map\)

V. Experiments

This section discusses the results of the experiments for our implementation of QAGen [2] and MyBenchmark [10]. Our implementation is written in Java 8. We have used MySQL 5.1.23 for the generation of the symbolic and final databases. The experiments were executed on a laptop computer with an Intel Core i7 2.20GHz CPU and 8GB memory. For the input queries, we have used TPC-H benchmark. This standard benchmark consists of a suite of business oriented ad-hoc queries. The values for the table sizes and the cardinality constraints are extracted from QAGen. In all the experiments, we select uniform distribution.

The experiments are organized as follows: 1) we evaluate the efficiency and the scalability of our implementation for QAGen, and 2) we perform the same evaluation for our implementation of MyBenchmark.

A. Efficiency and scalability of our QAGen implementation

The goal of this experiment is to evaluate 1) the efficiency of the symbolic query processing engine to impose the constraints from the input queries into the target database, 2) the scalability of this module using input queries with different cardinality constraints values, and 3) the running time of the data instantiator module to generate concrete data for databases of different sizes.

For this experiment, we use queries 1, 3 and 8 from TPC-H benchmark. These three queries allow us to evaluate all the cases for our relational algebra operations (e.g. the special case of the join operator that can be characterized as the...
Subset-sum problem). In QAGen, query 8 is used to evaluate the efficiency of the symbolic query processing. The paper shows the cardinality values in all the operations for this query to generate databases in different scales. We use these cardinalities to annotate queries 1, 3 and 8. For every query, we generate five query-aware databases in different scales (10M, 20M, 30M, 40M, 50M). Figure 8 and 9 depict the running time of the symbolic query processing and the data instantiator module respectively. The SQP phase is fast and its runtime is almost the same for the three queries. The first time we ran this experiment using Query 8, it took about 12 minutes to generate a database of 10M of size. This poor performance was due to the excessive number of accesses to the database (one for each getNext() method call). After implementing the optimization mentioned in Section IV, we have been able to reduce that number drastically. To process Query 8, the SQP engine performed hundreds of thousands of accesses to the database. Now only 18 accesses are executed (one per operator in the query). That allowed us to reduce those 12 minutes from the first execution to less than a minute.

DI phase also scales linearly. Nevertheless, it is much slower than SQP phase. In order to improve its performance, we implement the cache system described in QAGen. This system reduces the number of calls to the constraint solver. As we mentioned earlier, constraint solver calls are expensive. The cache system consists of storing values already instantiated in the databases for reuse. Although this cache system improves the running time of this phase saving calls to the constraint solver, a lot of accesses to the databases are still required.

To verify the quality of the query-aware databases, we pose the queries on every database and compared the actual cardinalities against the input cardinalities. Query 8 presents a deviation of 1 and 2 tuples for the 20M and 50M databases respectively. This deviation is the result of our approximate dynamic programming algorithm for the special case of the join operator. However, these deviations are considered acceptable. In the rest of the cases, the query-aware databases meet all the cardinality constraints.

B. Efficiency and scalability of our MyBenchmark implementation

The objective of this experiment is to evaluate 1) the efficiency of the symbolic database integrator to reduce the number of symbolic databases, and 2) the scalability of our integration algorithm. We use queries 1, 3, 5, 6, and 8 from TPC-H benchmark since TPC-W and TPC-C benchmarks used in MyBenchmark are no longer available. We scale down the cardinalities from QAGen to generate databases of 1M of data. We set the quality threshold to 50% for cardinalities in range [1, 100]. For example, the acceptable value for cardinality 20 is [10, 30].

To evaluate the efficiency of our symbolic database integrator, we run two experiments changing the order in which the queries are processed (ascending order and descending order). Table I shows the number of resulting databases and the relative error between the input cardinalities and the actual cardinalities when we pose every query in the database. The number of resulting databases is one single instance in both experiments. However, when processing the queries in ascending order, the quality of the database decreases more than if we process them in descending order. Query 5 and 8 have a relative error of 50% and 40% respectively for the first case against the 17% of relative error of Query 5 for the second case. This quality difference is because the integrator module does not consider reference constraints during the integration of two tables. These references are handled by the join operator during the symbolic execution of a query. The referenced values are assigned controlling the cardinality constraint. When we integrate two tables it is possible that the data distribution for one query affects the cardinality results of other queries. MyBenchmark mitigates this problem computing the integration plan that maximizes the quality of the database. Further research about this topic is part of our future plan.

To evaluate the scalability of our integration algorithm, we use Query 1 from Figure 2a. This is a very simple query that involves just one table. However, it is sufficient to test how our backtracking algorithm scales. We create graphs of different sizes from 1000 nodes up to 9000 nodes. Figure 10
shows the performance of our algorithm for this experiment. The algorithm runs fast for small graphs, growing in linear time for graphs less than 8000 nodes. However, when we create larger graphs, the algorithm grows very fast in time and space. To create the edges of a graph of 9000 nodes, the algorithm must compute more than 20 millions of edges, which is very expensive in terms of space. MyBenchmark reduces the number of nodes in the graph transforming the maximum satisfiable problem into a maximum flow problem in network optimization (see Trick 2). The incorporation of this optimization is also part of our future work.

VI. CONCLUSIONS

We explore the hardness of generating synthetic databases that comply with a set of user requirements (e.g. data distribution, test SQL queries). Although the data generation process requires solving multiple NP-Hard problems, we show it is possible to create synthetic query-aware databases in practice. To impose user requirements over databases, we implement the symbolic query processing technique described in QAGen [2]. This technique allows processing complex queries annotated with large cardinality constraints, and we show that it scales in linear time using TPC workloads. To reduce the number of final databases generated, we implement a simplified version of the integrator module described in MyBenchmark [10]. Our experiments show that our implementation is able to reduce the number of final databases from five to one database. However, further improvements are required to reduce the space and time complexity of our algorithm. Another improvement consists of maximizing the quality of the generated databases selecting the most adequate integration plan for every set of input queries.

TABLE I: Database quality processing TPC queries in different order

<table>
<thead>
<tr>
<th>Query</th>
<th>Cardinality</th>
<th>Ascending order</th>
<th>Descending order</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Input 15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Output 15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>Input 15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Output 15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>Input 20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Output 30</td>
<td>30 (50% relative error)</td>
<td>17 (15% relative error)</td>
</tr>
<tr>
<td>6</td>
<td>Input 35</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>Output 35</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>8</td>
<td>Input 10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Output 6</td>
<td>0 (40% relative error)</td>
<td>0 (40% relative error)</td>
</tr>
</tbody>
</table>

Number of databases 8 5 3 1

Fig. 10: Integration algorithm scalability for graphs of different sizes

REFERENCES


