Generating Databases for Query Workloads*

Eric Lo† Nick Cheng† Wing-Kai Hon‡
†Hong Kong Polytechnic University ‡National Tsing Hua University
†{ericlo, csutcheng}@comp.polyu.edu.hk ‡wkhon@cs.nthu.edu.tw

ABSTRACT

To evaluate the performance of database applications and DBMSs, we usually execute workloads of queries on generated databases of different sizes and measure the response time. This paper introduces MyBenchmark, an offline data generation tool that takes a set of queries as input and generates database instances for which the users can control the characteristics of the resulting workload. Applications of MyBenchmark include database testing, database application testing, and application-driven benchmarking. We present the architecture and the implementation algorithms of MyBenchmark. We also present the evaluation results of MyBenchmark using TPC workloads.

1. INTRODUCTION

Query performance is a key factor of a successful database (DB) application and DBMS. To evaluate the performance of DB applications and DBMSs, we usually execute workloads of queries on generated databases in different sizes and measure the response time.

This paper presents a workload-aware data generator, MyBenchmark. Given a database schema H and a set of queries, MyBenchmark allows users to generate databases in different sizes with the power to control not only the characteristics of the generated data (e.g., value distribution) but also the characteristics of the workload (e.g., cardinality of intermediate query operators). The applications of MyBenchmark include the following:

• **Testing DBMSs** Recent papers [5, 16, 4, 14] have pointed out that controlling the cardinalities of query operators in a test query is very useful in DBMS testing. For example, testers can study the performance of a hash-join implementation by varying the input and output cardinalities of the join operator [5]. Recent data generation technology has made some progress in this respect. QAGen [4] is an offline test database generator designed for this purpose. It takes a test case and a database schema H as input. A test case is a parameterized query Q with operators and base tables annotated with cardinality and data distribution constraints. The output of QAGen is a query-aware database D that conforms to H and a set of parameter values P. Executing query Q (with parameter values P) on D (denoted as $Q_P(D)$) guarantees that the constraints annotated on Q are satisfied. QAGen every time takes only one test case as input and generates an independent test database that is specific for that test case. To carry out a test of n test cases on a DBMS product, the test team needs to maintain n separate test databases, which require a prohibitively high storage cost [16] (imagine a test suite of 1000 test cases, where each test case demands a 10GB test database).

Differing from QAGen, MyBenchmark takes *a set of annotated parameterized queries* (or in this context, a set of DBMS test cases) as input, and generates *a minimal set of database instances* with the same query cardinality and data distribution assurance as QAGen does. As such, tests on DBMSs can be carried out more space efficiently.[1]

• **Stress testing database applications** Consider a DB application with n SQL queries. Developers of that application can use MyBenchmark to generate a variety of synthetic workloads to stress the application. For example, a developer may use MyBenchmark to generate a 1GB database that guarantees all the application queries return millions of rows.[2] This functionality allows the developers to test the functional and performance limits of their applications.[1]

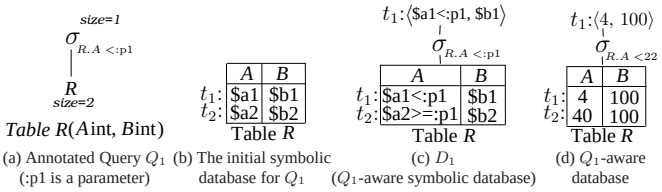
• **Application-driven benchmarking** Benchmarking requires the generation of benchmark databases. Existing benchmarks such as TPC benchmarks, although comprehensive, may not 100% reflect the performance of a DBMS with respect to an enterprise's environment because of the differences in the schemas between TPC benchmarks and the enterprise's DB applications. By using MyBenchmark, an enterprise is able to study the performance of a DBMS with respect to its own DB applications. Suppose a new start-up wishes to purchase a DBMS. The start-up may wish to know which DBMS (e.g., Oracle, SQL Server) performs the best for its application when dealing with one billion customer records and selective user queries. The start-up can use MyBenchmark to generate the relevant data and evaluate the DBMSs using its own set of database application queries. These application-specific benchmark results can complement the TPC benchmark results and provide supplementary information to the company when purchasing

*Research supported by grant PolyU 525009E from Hong Kong RGC.

Proceedings of the VLDB Endowment, Vol. 3, No. 1

[1] In case MyBenchmark generates more than one test database, we may use a database testing framework (e.g., DbUnit [1], HTpar [11]) to automatically assign the generated databases to the test queries.

[2] In this case, the developers need to specify only the output cardinalities of the final results and may leave the constraints of intermediate operators empty.



(a) Annotated Query Q_1 (:p1 is a parameter) (b) The initial symbolic database for Q_1 (c) D_1 (Q_1-aware symbolic database) (d) Q_1-aware database

Figure 1: Symbolic query processing (SQP)

its DBMS.

To the best of our knowledge, we are the first to study the generation of workload-aware data. Compared with the state-of-the-art (single)-query-aware data generation technology, workload-aware data generation is more general and has more applications, but is also more challenging. This paper contains our solution to the problem, including the architecture and algorithms of implementing MyBenchmark. This paper also contains the evaluation results of MyBenchmark using TPC workloads.

The rest of this paper is organized as follows. Section 2 covers the background and related work. Section 3 presents the architecture and algorithms of MyBenchmark. Section 4 summarizes the methodology of generating workload-aware data using MyBenchmark. Section 5 shows the experimental results. Section 6 concludes the paper with future research directions. The appendix contains some supplementary details and the proofs of the lemmas presented in the paper.

2. BACKGROUND AND RELATED WORK

Query-aware data generation was first studied by [15] and has received renewed attention in recent years. In [15], the authors studied the generation of test data that complies with functional dependencies for simple relational queries. In [3], the authors studied the generation of test data for functional testing database applications. The focus of [3] is to generate *minimal size* test databases for a *single* application query. In [17], the authors discussed the extraction of example data to facilitate dataflow (e.g., MapReduce) programming. However, it also focuses on getting the smallest amount of data as possible for the ease of human understanding.

We now give a brief background on QAGen. We refer readers to Appendix A or [4] for additional details. The QAGen system [4], the predecessor of MyBenchmark, is a query-aware test database generator that takes an annotated parameterized query Q and a database schema H as input. Each operator or base table in Q is annotated with a set of constraints (usually cardinality and data distribution). Figure 1a shows an annotated selection query Q_1 as an example. Q_1 specifies that table R should be populated with two tuples and the query should return one tuple (`:p1` is a parameter).[3] Figure 1d shows the output of QAGen for Q_1, which is a query-aware database D that conforms to H, and a set of parameter values P. Executing query Q (with parameter values P) on D guarantees that the constraints defined on Q are satisfied.

To process a query like the one in Figure 1a before the data is generated, QAGen introduces the concept of symbolic query processing (SQP). SQP starts with the population of a symbolic database (SDB) according to the sizes of the base tables specified in the annotated query (Figure 1b). Tuples in an SDB contain symbols rather than concrete values. During SQP, an operator evaluates the input tuples according to its own semantics and at the same time, it controls its output to its parent operator so that the parent operator can work on the right tuples. After symbolic query processing, the set of symbolic relations capture all the constraints defined on the input query (Figure 1c). In the final step, QAGen has a data instantiator to instantiate the symbolic tuples and the parameters and a query-aware database is generated (Figure 1d).

SQP is an advanced data generation technology that is much more complicated than traditional query-unaware data generation technology such as [10]. Therefore, query-aware data generation tools usually have a longer running time and run as an offline (background) process [14]. Nevertheless, the process can be easily parallelized using n machines to SQP n queries.

3. MYBENCHMARK

MyBenchmark uses the symbolic query processing technique developed in [4] as a building block. However, as we will show later, the generation of a *single* symbolic database for multiple queries is $\mathcal{NP}$-hard; thus, we do not restrict ourselves to find a single database instance D for all input queries. Instead, given a database schema H, a set of annotated queries $\mathcal{Q}= \{Q_1, Q_2, \ldots, Q_n\}$ (the operator(s) in Q_i are annotated with cardinality constraint(s) C_i), MyBenchmark generates m ($m \leq n$) databases $D_1, D_2, \ldots, D_m$ and m sets of parameter values $P_1, P_2, \ldots, P_m$, such that (1) all databases D_j ($1 \leq j \leq m$) conform to H, and (2) the resulting cardinalities C_i' of executing Q_i on one of the generated databases D_j, using the parameter values P_j, approximately meet C_i (the degree of approximation defined is based on the relative error between actual cardinalities and annotated cardinalities; details are in Section 3.2). Approximate cardinalities are sufficient for applications such as DBMS testing [5, 16] and database application testing [3]. Assume that a DBMS test engineer wants to use MyBenchmark to generate a workload with a 1GB database and ten application queries, in which one of the queries, Q_1, is annotated by the tester as a highly selective query that returns one row. In this case, a generated database that returns five rows for Q_1 is still very acceptable. As SQP controls the data distributions through the operator cardinalities, so we focus on the control of the operator cardinalities. Also, we put our focus on SPJ (select-project-join) queries in this paper.

Of course, if $m = n$, that essentially means MyBenchmark is the same as QAGen in which each query has to be executed on a separate generated database. Therefore, the goal of MyBenchmark is to minimize m, the number of generated databases, in best effort.

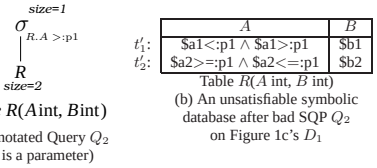
3.1 System Architecture

SQP was designed to generate n separate databases for the n input annotated queries. If SQP is carried out on a "processed" symbolic database, SQP will generate many symbolic tuples with *contradicting* constraints (as different queries may impose different constraints on the *same* symbolic tuple) and they will be unable to be instantiated with concrete values.

To illustrate, assume that we need to generate a database for two (annotated) application queries Q_1 and Q_2. Let Q_1 be the query given in Figure 1a; and Q_2 be a selection query in Figure 2a, which specifies that table R should have two tuples and the query should return one tuple.[4] Assume Q_1 is first symbolically processed by the SQP engine and we obtain the symbolic database D_1 (Figure 1c). If the second query Q_2 is directly processed on D_1, the selection operator of Q_2 may annotate the positive constraint [>:p1] to t_1 and the negative constraint [<=:p1] to t_2. That will result in an

[3]Each input query is practically formulated as a number of SQL statements and expected cardinality/distribution.

[4]In fact, Q_2 must annotate consistent constraints with Q_1 (e.g., two tuples for table R) or otherwise MyBenchmark will return an error to the user.



(a) Annotated Query Q_2 (:p1 is a parameter)

(b) An unsatisfiable symbolic database after bad SQP Q_2 on Figure 1c's D_1

Figure 2: Examples of SQP on a "processed" SDB

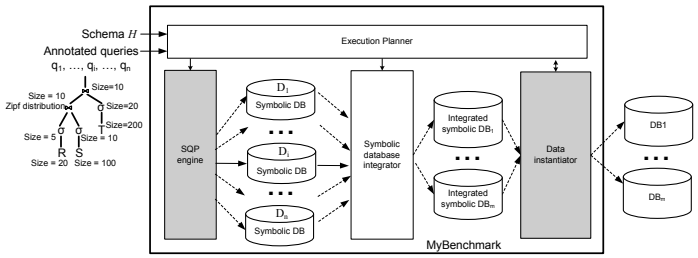


Figure 3: MyBenchmark architecture

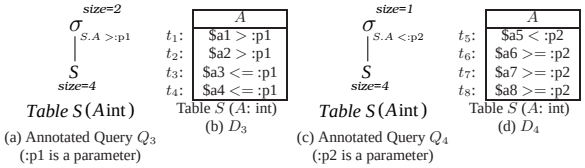
SDB (Figure 2b) in which tuple t'_1 is associated with a contradicting constraint [$a1<:p1 ∧ $a1>:p1].

Figure 3 shows our proposed architecture for MyBenchmark. To generate m databases for n annotated queries $Q_1, Q_2, \ldots, Q_n$, MyBenchmark first uses QAGen's SQP engine as a black-box component to process each annotated query separately (without data instantiation) and generates n symbolic databases $D_1, D_2, \ldots, D_n$. Each symbolic database D_i guarantees that $Q_i(D_i)$ satisfies the constraints annotated on Q_i. Then, a Symbolic Database Integrator is used to integrate the SDBs. The integration algorithms are designed to minimize the number of symbolic tuples with contradicting constraints (e.g., t'_1 in Figure 2b) and the number of generated databases. Finally, we use the Data Instantiator of QAGen to instantiate each integrated SDB with concrete values. The major advantage of this architecture is that we can fully utilize the capability of SQP in processing a variety of SQL queries. The Execution Planner is designed for integrating multiple SDBs and we defer its discussion until Section 3.3.

3.2 Symbolic Database Integration

We begin with the discussion of integrating two symbolic relations (with the same table definitions) that are separately generated by the SQP engine for two annotated queries. We discuss the integration of multiple symbolic relations in the end of this subsection and the integration of multiple symbolic databases in Section 3.3.

We use the annotated SQL queries Q_3 and Q_4 in Figures 4a and 4c as the running example. For ease of exposition, both Q_3 and Q_4 are simple selection queries posed on table S. Figures 4b and 4d show the corresponding symbolic databases D_3 and D_4 that are generated by the SQP engine for Q_3 and Q_4. When only two symbolic relations are involved, the major challenge for the symbolic data integrator is to minimize the number of symbolic tuples with contradicting constraints. In other words, the integrator cannot simply merge t_1 with t_5, (i.e., treating symbols $a1 and $a5 as the same symbol and joining the constraints of t_1 and t_5 together to get [$a1>:p1 ∧ $a1<:p2]), t_2 with t_6, t_3 with t_7, and t_4 with t_8. Such a naive integration would result in an integrated symbolic database $\bar{D}$ as shown in Figure 5a. The problem with $\bar{D}$ is that many symbolic tuples are contradicting with each other: t'_9 induces a *relationship* :p2 $>$:p1, but t'_{11} and t'_{12} induce a relationship :p2 $\leq$:p1. As such, the integration algorithms should be designed to



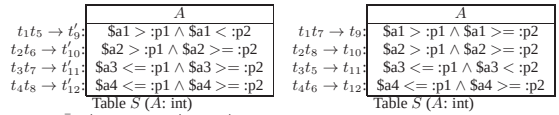
(a) Annotated Query Q_3 (:p1 is a parameter)

(b) D_3

(c) Annotated Query Q_4 (:p2 is a parameter)

(d) D_4

Figure 4: Examples for symbolic database integration



(a) $\bar{D}$ (t'_9 contradicts t'_{11} and t'_{12})

(b) Ideal integrated symbolic database

Figure 5: SDBs integrated from D_3 (Figure 4b) and D_4 (Figure 4d)

minimize the number of symbolic tuples with contradicting constraints in the integrated SDB. For example, Figure 5b shows an ideal symbolic database that is integrated from D_3 and D_4, and does not contain any tuples with contradicting constraints.

To integrate two symbolic relations S_i and S_j (where S_i and S_j share the same table definition), we model the problem as a graph problem.

DEFINITION 1. (CONSTRAINED NODE). A node n is *constrained* iff it is associated with a propositional formula, ϕ_n, composed of variables under a finite domain (SQL data types). □

DEFINITION 2. (SATISFIABLE EDGE). An edge $e(u, v)$ is *satisfiable* iff the conjunction of the propositional formula associated with constrained nodes u and v is satisfiable. That is, $\phi_u \wedge \phi_v$ is satisfiable. □

As an example, consider an edge $e(u, v)$ connecting two constrained nodes u and v. Assume u is associated with a propositional formula $x > p$ and v is associated with a propositional formula $x < p$, then e is not satisfiable. On the contrary, if u is associated with a propositional formula $x > p$ and v is associated with a propositional formula $y < 10$, then e is a satisfiable edge.

DEFINITION 3. (CONSTRAINED BIPARTITE GRAPH) A graph $G = (U, V, E)$ with node sets U and V and edge set E is a *constrained bipartite graph* (CBG) if G is a bipartite graph, all nodes in U and V are constrained nodes, and all edges in E are satisfiable edges. □

Now, we can model a symbolic tuple t_i (t_j) of symbolic relation S_i (S_j) as a constrained node u_i (v_j) in a CBG G. For each pair of tuples $t_i \in S_i$ and $t_j \in S_j$, if the conjunction (of the constraints) of t_i and t_j is satisfiable (i.e., no contradiction), we add a satisfiable edge $e(u_i, v_j)$ to G. As a result, the two symbolic relations in Figures 4b and 4d can be modeled as a constrained bipartite graph G_a shown in Figure 6a.[5] Now, we can model the integration of S_i and S_j as finding a *maximum satisfiable matching* of a CBG.

DEFINITION 4. (SATISFIABLE MATCHING) Given a constrained bipartite graph $G = (U, V, E)$, a matching M is *satisfiable* iff the conjunction of the propositional formulas associated with all constrained nodes in M is satisfiable. □

[5]Note that G_a is not necessarily *complete*.

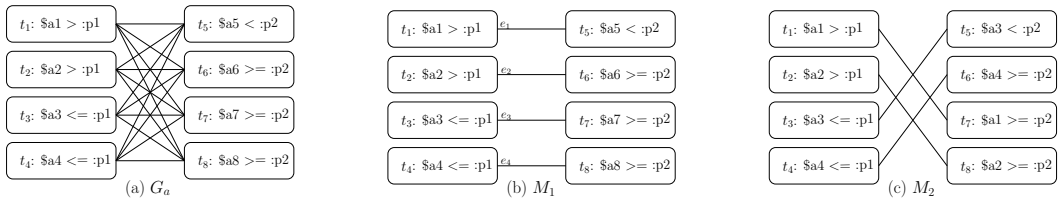


Figure 6: (a) A constrained bipartite graph G_a modeling the integration of databases D_3 and D_4 in Figure 4 (b) A maximum but not maximum satisfiable matching M_1 (c) A maximum satisfiable matching M_2

DEFINITION 5. (MAXIMUM SATISFIABLE MATCHING) Given a constrained bipartite graph $G = (U, V, E)$, a satisfiable matching M is *maximum satisfiable* iff the size of M is largest among all satisfiable matchings in G. □

The size of a maximum satisfiable matching (MSM) could be different from the size of a maximum matching. Figure 6b shows a maximum but not satisfiable matching M_1 of G_a. Edge e_1 in M_1 suggests that tuple t_1 of D_3 in Figure 4b should be integrated with tuple t_5 of D_4 shown in Figure 4d. Therefore, if the integration follows M_1, the resulting integrated database would become $\bar{D}$ in Figure 5a. On the other hand, if the integration follows M_2 (see Figure 6c), which is a maximum satisfiable matching of G_1, the resulting integrated database would become the ideal integrated symbolic databases shown in Figure 5b.

We cast the problem of finding an MSM of a constrained bipartite graph as a decision problem:

DEFINITION 6. (k-SAT-MATCH PROBLEM). Given a constrained bipartite graph $G = (U, V, E)$ and an input integer k, the decision problem k-SAT-MATCH is to answer if there is a satisfiable matching M of size that is at least k.

Searching a maximum matching from a bipartite graph can be done in polynomial time. However, searching a maximum satisfiable matching from a CBG is $\mathcal{NP}$-hard (proof in Appendix C.1). The main difficulty lies in the requirement of "satisfiability" among the induced relationships of variables at run-time (e.g., in Figure 6b, adding edge (t_1, t_5) to M_1 will induce a relationship that hinders adding edges (t_3, t_7) and (t_4, t_8) to M_1). This is also the main reason why applying SQP on a "processed" SDB online (mentioned in Section 3.1) is not a good idea. Nevertheless, we have developed many tricks to avoid the worst case in almost all circumstances. Specifically, we have developed a best-effort symbolic database integration algorithm that utilizes the special properties of SQP to reduce the search space. Our experiments show that SI practically solves the problem and scales well under a variety of inputs.

The Symbolic Database Integration Algorithm

The symbolic database integration algorithm (SI) solves the maximum satisfiable matching by separating the induced relationship problem and the maximum matching problem. The main idea is as follows. Given a constrained bipartite graph G as input, (1) it first identifies all the total-order relationships that can be induced by the satisfiable edges and puts them in a set $\mathcal{R}$. (2) For each possible subset R_i of $\mathcal{R}$, it constructs a new constrained bipartite graph G_i. G_i includes the edges that induce total-order relationship(s) in R_i and the edges that induce no total-order (edges that induce only partial-order relationships). (3) Find a maximum matching M_i for each constructed bipartite graph G_i. (4) Finally, for all the maximum matchings found, follow (any) one that has the maximum matching size to perform tuple integration.

Assume that SI takes G (Figure 6a) as input. As a first step, a total-order relationship r_1=[:p2 $>$:p1] induced by edges (t_1, t_5) and (t_2, t_5) and a total-order relationship r_2=[:p2 $\leq$:p1] induced by edges (t_3, t_6), (t_3, t_7), (t_3, t_8), (t_4, t_6), (t_4, t_7), and (t_4, t_8) are added to $\mathcal{R}$. Next, four constrained bipartite graphs G_1, G_2, G_3 and G_4 are constructed according to Step 2 above. Specifically, G_1 (shown in Figure 7a) includes the edges that induce the total-order r_1 and the edges that induce no total-order (e.g., (t_1, t_6)). G_2 (shown in Figure 7b) includes the edges that induce the total-order r_2 and the edges that induce no total-order. G_3 includes the edges that induce r_1 and r_2 and the edges that induce no total-order (G_3 is the same as the input graph). G_4 includes the edges that induce no relationships.[6]

By following the basic idea illustrated above, SI has to search maximum matchings for $2^{|\mathcal{R}|}$ constrained bipartite graphs. Although it looks a lot on the surface, $|\mathcal{R}|$ is actually a small number in practice. For example, in our experiments, the maximum values of $|\mathcal{R}|$ found in TPC-W and TPC-C workloads 13 and 24, respectively. Furthermore, we have incorporated four techniques into SI such that it actually visits only at most some tens CBGs in all our experiments. We have also devised an approximation version of SI that runs in linear time. However, our experiments show that the exact version of SI, in practice, scales well, and finds perfect matchings easily such that the most time consuming part is usually the preparation of SDBs using SQP. As such, we do not present the approximation solution here.

Trick 1. Pruning CBGs that are constructed from contradicting relationships The following lemma tells us that if a CBG G_i contains some *contradicting* relationships, SI can ignore G_i because there exists another CBG G_j with a larger MSM.

LEMMA 1. *Let r_i and r_j be two contradicting total-order relationships and let R_{ij}, R_i, and R_j be three relationship sets. Assume $\{r_i, r_j\} \in R_{ij}$, $R_i = R_{ij} - \{r_j\}$, and $R_j = R_{ij} - \{r_i\}$. Let G_{ij}, G_i, and G_j be the constrained bipartite graphs constructed from R_{ij}, R_i, and R_j, respectively. If M_{ij}, M_i, and M_j are maximum satisfiable matchings of G_{ij}, G_i, and G_j, respectively, then* $|M_{ij}| = \max(|M_i|, |M_j|)$.

The proof of Lemma 1 is in Appendix C.2. In our example, by Lemma 1, SI does not need to consider G_3 because r_1 and r_2 are contradicting and therefore the size of the MSM of G_3 would not be larger than the size of both the MSM of G_1 and the MSM of G_2.

Trick 2. Compressing the problem instances SI's efficiency can be further improved by *compressing* the symbolic tuples. For instance, in Figure 7a, tuples t_1 and t_2 are capturing the same selection predicate `S.A > :p1` of query Q_3. Therefore, they are compressed into a *single* node. A maximum matching problem is often

[6]We do not show G_4 here for space reasons.

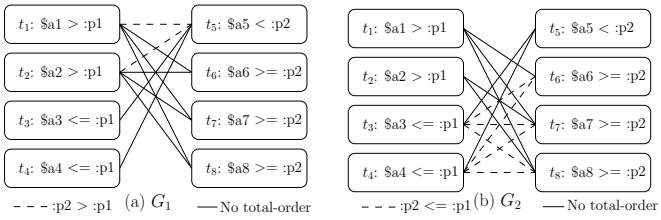


Figure 7: Examples for algorithm SI

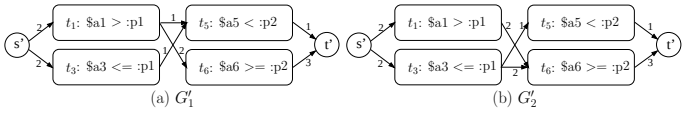


Figure 8: Flow networks

transformed into a maximum flow problem in network optimization [2, 12]. Here, SI compresses the input constrained bipartite graph $G = (U, V, E)$ into a constrained flow network $G' = (U', V', E')$:

i. (Build node sets) for every group of nodes N_u in U that captures the same predicate, add a node $n'_u \in U'$; similarly for V'.

ii. (Build edge set) add an edge between n'_u and n'_v if there was an edge (n_u, n_v) in G where $n_u \in N_u$ and $n_v \in N_v$.

iii. (Connecting source and sink) add an edge between source s' and $n'_{u'}$, and an edge between $n'_{v'}$ and sink t'.

iv. (Calculate edge capacities) for edges of the form (s', n'_u), the capacity is set to $|N_u|$; for edges of the form (n'_v, t'), the capacity is set to $|N_v|$; for edges of the form (n'_u, n'_v), the capacity is set to $min(|N_u|, |N_v|)$.

Figures 8a and 8b show the flow networks compressed from Figures 7a and 7b, respectively. We can see that the number of nodes is only half of the original constrained bipartite graph. In general, for select-project-join (SPJ) queries, the number of compressed nodes mainly depends on the number of predicates, not data size.

Trick 3. DFS and subset pruning Consider Figure 9, which is a more complicated constrained bipartite graph $G = (U, V, E)$, as an example. G represents an instance of integrating two symbolic relations after several rounds of integration, which often happens when multiple queries are input to the system (see Section 3.3 for details). For the time being, we focus on an MSM search for Figure 9.

In Figure 9, the set of edges induces the following set of total-order relationships $\mathcal{R}$={r_1=[:p1 > :p2], r_2=[:p2 ≥ :p1], r_3=[:p2 > :p1]}. For example, edge (t_1, t_7) induces a total order [:p1 > :p2] and edge (t_4, t_6) induces a total order [:p2 > :p1]. There

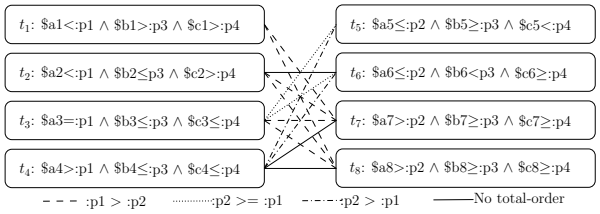


Figure 9: A CBG for a multiple-query integration instance.

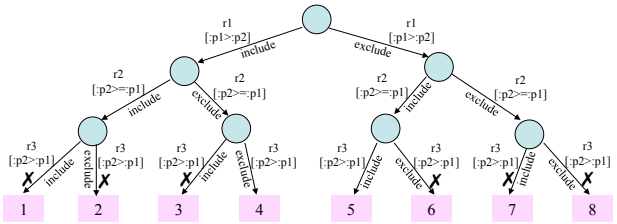


Figure 10: Search tree of SI. Lemma 1 prunes cases 1, 2 and 3. Lemma 2 prunes cases 6, 7, and 8.

are no total-orders induced from symbols connecting parameters :p3 and :p4. For the example in Figure 9, we can visualize its 2^3 cases (all possible subsets) as a search tree (Figure 10). The left branch of the search tree denotes the inclusion of a relationship and the right branch of the search tree denotes the exclusion of a relationship. As an example, leaf node 5 represents the case that we need to construct a constrained bipartite graph by including edges that induce relationship r_2=[:p2 ≥ :p1] (e.g., (t_3, t_5)), edges that induce relationship r_3=[:p2 > :p1] (e.g., (t_4, t_6)), and edges that induce no total-order (e.g., (t_4, t_8)). Looking at Figure 10, we see that Lemma 1 prunes cases 1, 2, and 3, as those cases include edges from contradicting relationships (r_1 contradicts both r_2 and r_3).

SI traverses the search tree in a depth-first manner because the order of node traversal helps prune the search space by the following lemma:

LEMMA 2. *Given two non-empty relationship subsets $\{R_i, R_j\} \in \mathcal{R}$, if $R_i \subseteq R_j$, the size of the MSM M_i, of the constrained bipartite graph constructed from R_i, must be less than or equal to the the size of the MSM M_j, of the constrained bipartite graph constructed from R_j (i.e., $|M_i| \leq |M_j|$).*

Lemma 2's proof is in Appendix C.3. By Lemma 2, SI can prune cases 6, 7, and 8 because the MSM obtained from these cases cannot be larger than the MSM obtained from case 5. Up to this point, SI needs to consider only cases 4 and 5.

Trick 4. Early Stopping Our goal is to find the largest MSM among all the possible CBGs. The last trick is, if SI finds a perfect satisfiable matching in a CBG, it can stop early. Although simple, experiments show that this trick is very useful since SI is often able to find a perfect satisfiable matching very early in the process.

Implementation, Pseudo-code, and Correctness The implementation details (e.g., choice of the maximum flow algorithm) and the pseudo-code of SI are in Appendix B. In Appendix C.4, we prove that algorithm SI returns an MSM of a CBG correctly.

Multiple Attributes and Multiple Tables Generalizing SI to handle multiple attributes is straightforward. In case a tuple contains multiple attributes, a single node is created for the conjunction of all the constraints in the attributes. In fact, Figure 9 is an example of said idea. Integrating two SDBs that contain more than one pair of symbolic relations is also straightforward. We can simply apply SI on every pair of overlapping symbolic relations.

3.3 Multiple Queries

We now discuss how to integrate multiple symbolic databases when each database is independently generated from a single input annotated query by SQP. Intuitively, to integrate n symbolic relations $S_1, S_2, \ldots, S_n$ (which share the same table definition and

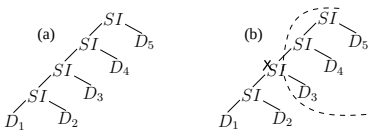


Figure 11: (a) An Integration Plan (b) An Execution Plan

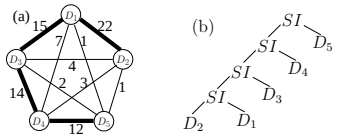


Figure 12: (a) A graph representing the (scaled-down) MSM sizes between pairs of input SDBs (bolded edges represent the MST) (b) An integration plan deduced from the MST.

are generated by SQP for n queries), we can model the problem as finding an MSM of a constrained n-partite graph; however, that problem is obviously too difficult to be solved while maintaining both a good running time and a good matching size. Therefore, our method of integrating multiple symbolic databases resembles the concept of joining.

We use $SI(D_i, D_j)$ to denote the integration of two SDBs D_i and D_j, and use D_{ij} to denote the resulting SDB. The integration of three SDBs, D_i, D_j, and D_k, can then be achieved by one of two integration plans, either $SI(SI(D_i, D_j), D_k)$ or $SI(D_i, SI(D_j, D_k))$. Figure 11a shows an integration plan of five SDBs. SDBs D_1 and D_2 are first integrated, then the resulting database D_{12} is then further integrated with D_3 and so on. An observation is that an SI operation is commutative, i.e., $SI(D_i, D_j) \equiv SI(D_j, D_i)$ in terms of matching size. Nevertheless, for performance reasons, SI is designed to return any one of the possible maximum satisfiable matchings (if multiple MSMs exist). Therefore, the MSM M_{ij} returned by $SI(D_i, D_j)$ may have a different set of matching edges with the MSM M_{ji} returned by $SI(D_j, D_i)$. Consequently, an SI operation is not associative, i.e., $SI(SI(D_i, D_j), D_k) \not\equiv SI(D_i, SI(D_j, D_k))$, in terms of running time and matching size. For instance, $SI(SI(D_i, D_j), D_k)$ may find a larger MSM than $SI(D_i, SI(D_j, D_k))$.

Recall that given the SDBs of n annotated queries, our goal is to integrate the n SDBs into as few databases as possible. As the MSM returned by an SI operation may not be a perfect matching, the size of the MSM may get smaller and smaller when the integration goes up to the root. In order to ensure the matching size, or the *quality*, of an integrated database at a particular level of integration is acceptable, MyBenchmark stops integrating two SDBs when the quality of an SI operation drops below a user-defined-threshold. Since the size of an MSM is not readily known to the users, we define the **quality** threshold (from the user perspective) as the relative error between the annotated cardinality and the actual cardinality (obtained by posing the query on the generated data). Consider Figure 11a again. Assume that after $SI(D_1, D_2)$, $SI(D_{12}, D_3)$ results in a database D_{123} in which posing a query (e.g., Q_2) on it finds some query operator with relative error exceeding the threshold. Then, MyBenchmark will not further integrate D_{123} with D_4. Instead, it discards D_{123} and integrates D_3 with D_4 and so on (see Figure 11b). In the example, two databases D_{12} and D_{345} are generated to serve five queries.

Determining a good integration plan

Since there is an exponential number of possible integration plans, deducing an optimal one that returns a minimum set of databases, which have the lowest error, is a challenging problem. In fact, it is as hard as finding the optimal joining plan [7], which is $\mathcal{NP}$-hard (see Appendix C.5). Traditional query optimization uses heuristics and estimation to solve the join plan selection problem. Our solution borrows ideas from there. Specifically, in traditional query optimization, we usually pre-build certain summaries (e.g., histograms) on the data and exploit those to estimate the best plan using some efficient algorithm. For MyBenchmark, we pre-build a summary about the quality of some core SI operations and exploit that to estimate the best plan using a simple graph algorithm. More specifically, our approach is to first pre-build a summary about the sizes of the MSM between every pair of input SDBs. To obtain the size of the MSM between a pair of SDBs D_i and D_j, we have to carry out $SI(D_i, D_j)$. Given n annotated queries (thus n SDBs), we have to execute C_2^n SI operations (if $SI(D_i, D_j)$ has no common table, that SI is skipped and the MSM between D_i and D_j is set to the largest possible integer). To optimize this process, we first scale down the the cardinalities requirements of the input queries (e.g., from generating 1GB data to 1MB data) by the test case generation tool in [14]. For example, the input query in Figure 4a can be automatically scaled-down to have table S annotated with two tuples and the output annotated with one tuple (the tool will make sure the scaling is meaningful and in proportion). This scale down optimization is built upon the observation that (1) the number of total order relationships and more importantly (2) the ratio between MSM size and the CBG size depend on the characteristics of the input queries (e.g., the selection predicates) but not the size of the databases to be generated. Thus, there would be no difference in (1), (2), and thus the number of resulting databases between generating 1MB and 1GB data (our experiments confirmed this). However, there would be a significant time difference between the two. More specifically, the running time of an SI operation mainly consists of: (T1) scanning the SDBs and constructing the flow network, (T2) running the maximum flow algorithm, (T3) loading and merging tuples according to the MSM and inserting them into a new SDB, and (T4) the algorithmic overhead (e.g., checking contradicting total-orders). Using the four SI tricks, experiments show that (T1) often is the most time consuming step because QAGen (and thus MyBenchmark) store the symbolic/instantiated tuples in a PostgreSQL (the data is usually too large to fit in memory). Therefore, much time is spent on the overhead (e.g., JDBC) of reading symbolic tuples from the database. By running SI operations on the scaled-down SDBs instead, we can obtain the summary about the (proportionally scaled-down) sizes of the MSM between every pair of SDBs (at the leaf level) more efficiently. As a note, this summary can be obtained efficiently because it is independent of the annotated data size and operator cardinalities.

The summary obtained is represented as a graph. In the graph, a node denotes an SDB, an edge denotes an SI operation between a pair of SDBs D_i and D_j, and the edge weight denotes the MSM size between D_i and D_j. Figure 12a shows an example of such a graph for the five SDBs. Recall that, one additional database is required whenever the quality of the resulting instantiated database drops below the user-threshold. Actually, that is directly related to the size of the MSM obtained from each SI operation. Since our goal is to minimize the number of generated databases, the best plan should be the one that maximizes the MSM size of each SI operation. Therefore, we suggest that the best integration plan should be derived from the Maximum Spanning Tree (MST) of the graph. In Figure 12a, the MST of the graph is highlighted. Based

on the MST, the suggested integration plan (Figure 12b) follows a decreasingly ordered, by the edge weight (the MSM size), sequence of the MST, i.e., $SI(SI(SI(SI(D_2, D_1), D_3), D_4), D_5)$.

The plan selection is taken care by the Execution Planner (see Figure 3) of the system. Similar to any query plan selection algorithm, our approach is also based on heuristics and estimation, which may not find the optimal plan. Nevertheless, experimental results about this approach are quite encouraging. In our experiments on TPC-C and TPC-W workloads, the plans suggested by our method successfully integrate all databases into only two databases.

4. SUMMARY OF THE METHODOLOGY

Overall, the execution of MyBenchmark is composed of two steps:

Step $\mathcal{A}$. Finding a good integration plan. This involves:
$\mathcal{A}_1$) Scale down the cardinalities in the input queries. This is done by the tool in [14] using negligible time.
$\mathcal{A}_2$) SQP the scaled down input queries to get the small SDBs. This step is done *once* by the SQP engine in [4] for each input query.[7]
$\mathcal{A}_3$) Build a summary (graph) of MSM size by running SI on every pair of small SDBs.
$\mathcal{A}_4$) Suggest a plan P by finding a Maximum Spanning Tree from the graph. We can use any Minimum Spanning Tree algorithm. In our implementation, we used Kruskal's algorithm [13], which runs in $\mathcal{O}(n^2 \log\, n)$ time. As n is generally a small number for typical database applications (e.g., a TPC-W implementation has only about 20 parameterized queries), this step runs very fast. □

Step $\mathcal{B}$. Executing P in the original scale. This involves:
$\mathcal{B}_1$) SQP the input queries to get the SDBs in original scale. This step is done *once* by the SQP engine in [4] for each input query.[7]
$\mathcal{B}_2$) Run SI operations according to P.
a) After each SI, instantiate the resulting SDB.
b) Pose the processed queries on the resulting database to check the quality, i.e., the relative error between the actual cardinality and the annotated cardinality of each operator. Add a new database if necessary.

5. EXPERIMENTS

We have carried out experiments on MyBenchmark using workloads from TPC-W and TPC-C benchmarks. The implementation and experimental setings are in Appendix D.1. In all experiments, we exclude IUD (INSERT, UPDATE, and DELETE) SQL queries. We also exclude "independent" queries that share no common tables with the others (e.g., a SELECT query that accesses a table X is removed from consideration if no other SELECT queries also access X). That is because the symbolic databases generated for independent queries can be "perfectly integrated" with other SDBs without any effort.

We characterize the efficiency of MyBenchmark based on the items listed in Section 4. The quality of the generated databases is characterized by the error between the annotated and actual cardinalities of all queries and their sub-queries.

Experimental Result: TPC-W Benchmark

The TPC-W benchmark models a typical web-commerce database application. We downloaded an open-source implementation of TPC-W from http://www.ece.wisc.edu/∼pharm/tpcw.shtml. For space reasons, we refer readers to the TPC-W specification for the details of the queries (we name the TPC-W queries according to their appearance order in the specification). After removing IUD and independent queries, 15 TPC-W queries remained. The expected cardinalities annotated on the operators of the queries are specified according to the actual cardinalities obtained by running the queries on the TPC-W data with scale factor 1.0 (they are scaled-down to become a scale factor of 0.1 during the plan search process). The breakdown of the whole plan search process is as follows (rows in gray mean they are not the implementations of this paper):

Item	Description / Sub-item	Time
$\mathcal{A}_1$	scale-down the input queries	<1s
$\mathcal{A}_2$	SQP all down-scaled queries once [7]	6min
$\mathcal{A}_3$	Build a summary (graph) of MSM size	180s
$\mathcal{A}_4$	Suggest a plan P by finding a MST from the graph	0.5s
	$\sum_{\mathcal{A}}$	9min

Running SQP on all scaled-down input queries to generate the small SDBs ($\mathcal{A}_2$) is more time consuming than the other parts ($\mathcal{A}_1$, $\mathcal{A}_3$, $\mathcal{A}_4$). This makes sense because query-aware data generation is much more advanced than traditional query-unaware data generation technology and thus requires time to process symbolic data. The time spent on running SI on C_2^{15} pairs of SDBs ($\mathcal{A}_3$) is 180s.

The following table shows the experimental result about step $\mathcal{B}$, i.e., executing the good integration plan in scale 1.0:

Item	Description/Sub-item	Total Time Spent
$\mathcal{B}_1$	SQP all queries in original scale [7]	53min
$\mathcal{B}_2$	Follow P to run SI operations	6min
	(a) After each SI, instantiate the resulting SDB	46min
	(b) Pose all the queries on the resulting database to check quality	0.26s
	$\sum_{\mathcal{B}}$	1hr46min

The overall running time, $\sum_{\mathcal{A}}$ + $\sum_{\mathcal{B}}$, is 1 hour 55 minutes. By following the suggested plan, two databases D_d and D_n were generated. When posing the original queries on the generated database, all queries obtain *exact* cardinalities as annotated in the input.

We now study the efficiency of SI and the pruning effectiveness of the tricks used by SI. The efficiency of the SI algorithm can be studied through the four items (T1 to T4) we mentioned in Section 3.3. The effectiveness of an SI can be characterized by (i) the *compression ratio*: the total number of tuples (E_1) vs. the total number of nodes in the flow network (E_2); and (ii) the *pruning effectiveness*: the number of all possible cases (E_3), the number of cases actually examined (E_4), and the number of cases pruned by Lemmas 1 and 2 (E_5). E_3, the number of all possible cases, can also be regarded as the performance of a "baseline" solution in which no tricks are used and can be used for comparisons. In this experiment, we used the best plan found in step $\mathcal{A}$ and measured the total values of the aforementioned items of all executed SI operations.

Figure 14 presents the time breakdown of all executed SI operations in different scale factors (1, 10, and 100). T1 is very time consuming because it reads a large number of tuples from the SDBs. Thanks to compression, the number of nodes (E2) is significantly smaller than the original data (E1). Thus, the maximum flow running time (T2) and the time to integrate and instantiate the compressed tuples (T3) are small. Most SI operations involved very few unique relationships and the number of unique relationships is the same across different scales. The $SI(D_j, D_9)$ operation involved 13 distinct relationships and thus the "baseline" solution needed to examine $2^{13} = 8192$ possible cases (E3). However, with our tricks, 7642 cases (E5) were actually pruned and the SI operation actually examined only 8 CBGs (E4) before a perfect matching was found and stopped early. In some SI operations (e.g., $SI(D_{11}, D_{10})$), there were no common tables and SI thus

[7] SQP and data instantiation are not our focus as long as they scale.

Num. of Queries	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sum_{\mathcal{A}}$ (in seconds)	24	50	78	107	139	173	209	247	287	329	373	419	467	517	569
$\sum_{\mathcal{B}}$ (in seconds)	416	634	1050	1466	1883	2299	2716	3132	3549	3865	4382	4798	5214	5631	6377
Num. of DB	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2

Figure 13: Scaling up the number of TPC-W queries

Running time breakdown of all SIs executed in part (B)	SF=1.0	SF=10.0	SF=100
T1 Constructing flow networks	122s	608s	5956s
T2 Maximum flow algorithm	0.383s	0.234s	0.255s
T3 Merging tuples & inserting them into SDBs	0.064	0.1s	0.111s
T4 Algorithmic work	251s	381s	419s
$\sum_{SI}$	374s	989s	6376s

Effectiveness of SI algorithm		
E1. Total num. of nodes in all SDBs	134K	1.2M	12M
E2. Total num. of nodes in all flow networks	56	72	67
	—the below is same for all scales—	
Table	customer	item
Pruning Effectiveness	E3 / E4 / E5	E3 / E4 / E5
$SI(D_{11}, D_{10}) \rightarrow D_a$	0 / 0 / 0	0 / 0 / 0
$SI(D_a, D_{14}) \rightarrow D_b$	0 / 0 / 0	0 / 0 / 0
$SI(D_b, D_{13}) \rightarrow D_c$	8 / 1 / 3	0 / 0 / 0
$SI(D_c, D_4) \rightarrow D_d$	0 / 0 / 0	0 / 0 / 0
$SI(D_d, D_{13}) \rightarrow D_e$	0 / 0 / 0	0 / 0 / 0
$SI(D_6, D_{15}) \rightarrow D_f$	0 / 0 / 0	0 / 0 / 0
$SI(D_f, D_8) \rightarrow D_g$	0 / 0 / 0	0 / 0 / 0
$SI(D_3, D_2) \rightarrow D_h$	0 / 0 / 0	128 / 4 / 124
$SI(D_h, D_7) \rightarrow D_i$	1 /1 / 0	0 / 0 / 0
$SI(D_i, D_3) \rightarrow D_j$	8 /1 / 4	0 / 0 / 0
$SI(D_j, D_9) \rightarrow D_k$	0 / 0 / 0	8192 / 8 / 7642
$SI(D_k, D_1) \rightarrow D_l$	32 / 1 / 15	0 / 0 / 0
$SI(D_l, D_{12}) \rightarrow D_m$	1024 / 1 / 512	0 / 0 / 0
$SI(D_m, D_5) \rightarrow D_n$	0 / 0 / 0	4096 / 24 / 3872

Figure 14: Details of SI algorithm (TPC-W)

examined 0 cases. The overhead (T4) (e.g., checking contradicting relationship) is relatively less significant when compared with T1. Overall, we can see that the SI algorithm scales linearly with the size of the generated data.

To study the performance of the overall methodology with respect to workloads of different sizes, we carried out an experiment that varies the number of annotated queries in the input. Table 13 summarizes the results (time is in seconds). The running time roughly scales linearly to the number of input queries. After we processed the 6-th query, the quality of the generated data was below threshold and thus one new database was added. All the generated databases are perfect (i.e,. no error).

We remark that experiments of this kind can only be carried out by adding real queries. Other kinds of controlled experiments may not be applicable. Specifically, the number of queries cannot be scaled up even higher by using randomly generated queries because they often return empty results (i.e., cardinality equals 0 in the output operator). Also, it is difficult to control the number of unique relationships in the workload because that depends on the query semantic. Nevertheless, as TPC benchmarks are simulating realistic workloads, we believe that the number of queries we used is enough to reflect realistic applications.

Other results and discussion. For space reasons, we put the experimental result of using TPC-C workload in the Appendix. Overall, we see that MyBenchmark successfully minimizes the number of generated databases. The running time scales linearly to the data size and the number of input queries. Nevertheless, the number of input queries has some impact on the number of generated databases. For all the experiments that we have conducted, we have executed thousands to millions of extra randomly generated integration plans (using a cluster of machines) to get a picture of how the best plans for each workload could be. We found that the integration plans suggested by our method have the same number of generated databases (and 0 error) as the best plan we could find in millions of plans, except in the experiment using TPC-C workload, where we found a plan that integrated all databases into one with little error, which is better than the one suggested by our method. As a future work, we will study the better plans found in those random trials to further improve our integration plan searching method.

6. CONCLUSION

MyBenchmark is a workload-aware data generator that takes as input a set of queries and generates database instances for which the users can control the characteristics of the resulting workload. Applications of MyBenchmark include database testing, database application testing, and application-driven benchmarking. Although the whole data generation process requires solving several difficult problems, our experiments show that our proposed methods are able to practically solve them. Our future work will focus on further improving the integration plan search methods.

7. REFERENCES

[1] Dbunit. http://www.dbunit.org.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[3] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *ICDE*, 2007.

[4] C. Binnig, D. Kossmann, E. Lo, and M. T. Ozsu. QAGen: Generating Query Aware Test Databases. In *SIGMOD*, 2007.

[5] N. Bruno, S. Chaudhuri, and D. Thomas. Generating Queries with Cardinality Constraints for DBMS Testing. *TKDE*, 2006.

[6] B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[7] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *ICDT*, 1995.

[8] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *CAV*, pages 296–300, 2005.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1990.

[10] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, 1994.

[11] F. Haftmann, D. Kossmann, and E. Lo. A framework for efficient regression tests on database applications. *VLDB Journal*, 16(1):145–164, 2007.

[12] A. Kini, S. Shankar, J. F. Naughton, and D. J. DeWitt. Database support for matching: limitations and opportunities. In *SIGMOD Conference*, 2006.

[13] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[14] E. Lo, C. Binnig, D. Kossmann, M. T. Ozsu, and W.-K. Hon. A Framework for Testing DBMS Features. *VLDB Journal*, 19(2):203–230, 2010.

[15] H. Mannila and K.-J. Räihä. Test data for relational queries. In *PODS*, pages 217–223, 1986.

[16] C. Mishra, N. Koudas, and C. Zuzarte. Generating targeted queries for database testing. In *SIGMOD*, pages 499–510, 2008.

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

APPENDIX

A. BACKGROUND OF QAGEN

The QAGen system [4] is the predecessor of MyBenchmark. QAGen is a query-aware test database generator that takes as input an annotated parameterized query Q and a database schema H as input. Each operator or base table in Q is annotated with a set of constraints (usually cardinality and data distribution). Figure 1a shows an annotated selection query Q_1 as an example. The SQL statement of Q_1 is `SELECT A FROM R WHERE R.A < :p1`. Q_1 specifies that table R should be populated with two tuples and the query should return one tuple (`:p1` is a parameter). The output of QAGen is a query-aware database D that conforms to H, and a set of parameter values P. Executing query Q (with parameter values P) on D guarantees that the constraints defined on Q are satisfied.

As a means to process a query like the one in Figure 1a before the data is generated, QAGen introduces the concept of symbolic query processing (SQP). In SQP, each operator is implemented as an iterator with methods *open()*, *getNext()*, and *close()*. SQP starts with the population of a symbolic database (SDB) according to the sizes of the base tables specified in the annotated query. Figure 1b shows the SDB initialized for query Q_1. A symbolic database consists of a number of symbolic relations. A symbolic relation is a collection of symbolic tuples. Inside each symbolic tuple, the values are initially represented by symbols rather than by concrete values. For instance, tuple t_1 in Figure 1b is a symbolic tuple of symbolic relation R and symbol $a1 represents any value under the domain of attribute A.

Since a symbolic database provides an abstract representation for concrete data, SQP can control the output of each operator in accordance with the user-defined constraints. Specifically, an operator in SQP evaluates the input tuples according to its own semantics. It manipulates the symbols in each input tuple in order to reflect the constraints defined on the operator. At the same time, it controls its output to its parent operator so that the parent operator can work on the right tuples. Continuing with the example in Figure 1b, when the *getNext()* method of the selection operator $\sigma_{R.A<:p1}$ is first invoked, it reads tuple t_1 from R, annotates a "positive" constraint [<:p1] (i.e., the selection predicate) to symbol $a1 and returns tuple ⟨$a1<:p1, $b1⟩ to its parent. When the *getNext()* method of the selection operator is invoked a second time, the selection operator reads the next tuple t_2 from R, and annotates a "negative" constraint [>=:p1] (i.e., the negation of the selection predicate) to symbol $a2. However, this time it does *not* return t_2 to its parent because the cardinality constraint (1 tuple) is already satisfied. After symbolic query processing, the set of symbolic relations capture all the constraints defined on the input query (see Figure 1c). In the final step, a constraint solver is used to instantiate the symbolic tuples and the parameters with concrete values.[8] Figure 1d shows the instantiated table R and we can see that executing Q_1 on R (with :p1=22) would get exactly one tuple as defined by the user. In SQP, joins and groupings are implemented by symbol replacements. For example, if a group-by query is annotated to return 1 group from table R, the same symbolic relation in Figure 1b will be initialized but the grouping operator will replace symbol $a2 with $a1 during *getNext()* on t_2. Since both tuples t_1 and t_2 contain $a1 in their attribute A, the data instantiator will instantiate them with the same concrete value. In SQP, the data distribution constraints are controlled by the cardinalities.

QAGen is mainly composed of three components: a Query Analyzer, a Symbolic Query Processing (SQP) Engine, and a Data Instantiator. The Query Analyzer is used to parse annotated-queries and determine the cardinality or the data distribution if they are not specified on some query operator. The SQP Engine is used to symbolically process the query and the Data Instantiator uses an external constraint solver called Cogent [8] to instantiate the processed SDBs and the parameters with real values according to the user-given data distributions. The SQP Engine includes the SQP implementations of most SQL operators including selection, projection, join, grouping, and aggregation. QAGen is thus able to generate databases for a variety of SQL queries.

[8] A constraint solver takes as input a constraint formula and returns an instantiation on each variable as output. E.g., if an input constraint formula is 40<$a1+$b1<100, a constraint solver may return $a1=55, $b1=11 (or any other correct instantiation) as output. Although the constraint satisfaction problem on a finite domain is $\mathcal{NP}$-complete, there are many best-effort constraint solvers that can practically solve many forms of constraints.

B. IMPLEMENTATION AND PSEUDO-CODE OF SI

Algorithm 1 presents the pseudo-code of SI. We have gone through Steps (1) to (3) in the main discussion. As the relationships are unweighted, so the search tree is constructed randomly. In terms of implementation, Steps 4(a) and 4(b) are merged so that we construct the flow network from the symbolic relations directly. To implement Step 4(c), we use a push-relabel maximum flow algorithm with complexity $\mathcal{O}(n^3)$ [6] (n is the number of nodes in the flow network). To implement Step 4(d), for each edge of the form (n'_u, n'_v) in the network flow G'_i with flow value f, SI matches f members of N_u to f members of N_v. Finally, in Step 5, SI follows the largest maximum satisfiable matching that it has found to perform tuple merging.

Algorithm 1 SI

(1) Identifies all the total-order relationships that can be induced by the satisfiable edges and puts them in a set $\mathcal{R}$.
(2) Construct a search tree T for each subset R_i of $\mathcal{R}$ that
  (i) contains no contradicting relationships and
  (ii) R_i is not a subset of another subset R_j.
(3) Initialize MAX-MSM=null to store the largest MSM discovered so far.
(4) Visit the search tree T in a depth-first order.
  (a) construct a new constrained bipartite graph G_i which includes
    (i) the edges that induce the relationships in R_i appear as a left branch of a node (inclusion); and
    (ii) the edges that induce no total-order (edges that induce only partial-order relationships);
  (b) transform G_i into its flow network counterpart G'_i;
  (c) find a maximum flow M'_i from G'_i by invoking a maximum flow algorithm;
  (d) transform the resulting maximum flow into maximum matching MSM;
  if a perfect satisfiable matching M is found, stop searching.
  if |MSM| > size-of MAX-MSM
    set MAX-MSM= MSM
(5) Follow MAX-MSM to perform the integration.

C. PROOFS

C.1 Proof of k-SAT-MATCH is $\mathcal{NP}$-complete.

THEOREM 1. *Problem k-SAT-MATCH is $\mathcal{NP}$-complete.*

We begin with proving k-SAT-MATCH is in $\mathcal{NP}$, and further show that it is $\mathcal{NP}$-hard by a reduction from the $\mathcal{NP}$-complete problem known as X3C (Exact Cover by 3-set).

COROLLARY 1. *k-SAT-MATCH is in $\mathcal{NP}$.*

PROOF. Each "yes" instance has a polynomial-size proof, which consists of the set of edges in the matching, and the set of values for each variable. Thus, each "yes" instance can be verified in polynomial time. □

COROLLARY 2. *k-SAT-MATCH is $\mathcal{NP}$-hard.*

PROOF. Obviously, if we solely focus on the constraint satisfaction problem (i.e., the condition on satisfiability required in Definition 4), k-SAT-MATCH is definitely $\mathcal{NP}$-hard. However, as we want to show the difficulty of the matching problem itself (e.g., adding an edge to the matching set will induce some relationships that hinder the matching of the other nodes), we assume here the constraint satisfaction step is at no cost.

We are going to reduce X3C (Exact Cover by 3-Set) to the k-SAT-MATCH problem. The X3C problem [9] takes as input a set of elements $\mathcal{S}$= $\{S_1, S_2, \ldots, S_{3n}\}$ and a collection of 3-element set $\mathcal{C}$= $\{C_1, C_2, ..., C_m\}$ and asks whether there is a sub-collection of $\mathcal{C}$, whose size is n, such that it exactly covers all elements of $\mathcal{S}$. The reduction is to construct a constrained bipartite graph $G = (U, V, E)$ as follows.

1. For each 3-element set $C_i = \{S_j, S_k, S_\ell\}$, insert 3 constrained nodes $u_{i,j}$, $u_{i,k}$, and $u_{i,\ell}$ to constrained node set U. The propositional formulas that are associated with $u_{i,j}$, $u_{i,k}$, and $u_{i,\ell}$ would be [\$$a_j \leq w_i$], [\$$a_k \leq w_i$] and [\$$a_\ell \leq w_i$], respectively (\$$a_j$, \$$a_k$, \$$a_\ell$ are symbols and w_i is any unique value).
2. For each element S_j, insert a constrained node v_j to constrained node set V. The propositional formula that is associated with v_i would be [\$$b_j \geq w$] (value w would be the same for all elements).
3. Connect the nodes in U and V if they are created from the same element S_j.

   For instance, assume a 3-element set $C_2 = \{S_4, S_5, S_6\}$ has inserted 3 nodes $u_{2,4}$, $u_{2,5}$, and $u_{2,6}$ to U in Step 1 and element S_4 has inserted a node v_4 into V in Step 2. Then, nodes $u_{2,4}$ and v_4 should be connected as both of them are created from element S_4.
4. For each 3-element set C_i, insert a node u_{C_i} with propositional formula [\$$c_i > w_i$] to U and insert a node v_{C_i} with propositional formula [\$$d_i \leq w$] to V and connect the two nodes with an edge.

The rest of the proof will establish:

PROPOSITION 1. *There is an exact cover of $\mathcal{S}$ if and only if the size of maximum satisfiable matching of G is exactly $3n+(m-n)$.*

Firstly, if the node $u_{i,j}$ appears in the MSM, it must be matched with the node v_j, so that it will induce the total-order relationship $w_i \geq w$. On the other hand, if u_{C_i} appears in the MSM, it must be matched with will v_{C_i}, so that it will induce the total-order relationship $w > w_i$. Thus, if either $u_{i,j}$, $u_{i,k}$, or $u_{i,\ell}$ appear in the MSM, we cannot have u_{C_i} in the MSM at the same time.

Suppose we denote z to be the number of i's such that $u_{i,j}$, $u_{i,k}$, or $u_{i,\ell}$ appear in the MSM. Then, the size of MSM is at most $3z + (m - z)$, which in turn is at most $3n + (m - n)$ since $z \leq n$.

The "only-if" direction. Next, suppose there is an exact cover of $\mathcal{S}$. In that case, let $C_{i_1}, C_{i_2}, \ldots, C_{i_n}$ be the 3-sets such that they exactly cover $\mathcal{S}$. This implies the elements in these 3-sets must be distinct from each other. Then, consider the following matching in G:

1. For each $i \in \{i_1, i_2, \ldots, i_n\}$, the corresponding nodes of C_{i_t}, i.e., $u_{i_t,j}, u_{i_t,k}, u_{i_t,\ell}$, are matched to v_j, v_k, and v_ℓ, respectively.
2. For each $i \notin \{i_1, i_2, \ldots, i_n\}$, u_{C_i} is matched to v_{C_i}.

The above matching is also satisfiable because the edges induce total-order relationships of the form $w_i \geq w$ when $i \in \{i_1, i_2, \ldots, i_n\}$, and of the form $w_i < w$ for other choice of i. Thus, all edges can be satisfied simultaneously. Finally, it is easy to check that the above matching has $3n + (m - n)$ edges, so that it is a maximum satisfiable matching.

The "if" direction. If the size of MSM is exactly $3n+(m-n)$, we claim that z, which is the number of i's such that $u_{i,j}$, $u_{i,k}$, or $u_{i,\ell}$ appear in the MSM, must be exactly n; in addition, for each such i, all $u_{i,j}, u_{i,k}, u_{i,\ell}$ must appear in the matching. If this claim is true, it will immediately imply the corresponding 3-sets C_i's (in total n of them) will cover exactly $\mathcal{S}$.

Now, it remains to prove the claim. We first show that $z = n$. If $z < n$, then the matching can contain at most $3z$ edges connecting some $u_{i,r}$ with v_r, and at most $m - z$ edges connecting some u_{C_s} with v_{C_s}, so that the number of edges is at most $3z + (m - z)$, which is less than $3n + (m - n)$. On the other hand, if $z > n$, then the matching can contain at most $3n$ edges connecting some $u_{i,r}$ with v_r (because v_r is limited), and at most $m-z$ edges connecting some u_{C_s} with v_{C_s}, so that the number of edges is at most $3n + (m - z)$, which again is less than $3n + (m - n)$. Thus, if the size of MSM is $3n + (m - n)$, we must have $z = n$.

Given $z = n$, there are at most $m - n$ edges connecting u_{C_s} with v_{C_s}. Thus, at least $3n$ edges must be connecting some $u_{i,r}$ with v_r. However, since there are only n values of i with $u_{i,j}$, $u_{i,k}$, or $u_{i,\ell}$ appear in the MSM, the previous statement is possible unless for each such i, all $u_{i,j}, u_{i,k}, u_{i,\ell}$ appear in the matching. Thus, the proof of the claim completes, and so do the proofs of the Proposition 1 and Corollary 2.

C.2 Proof of Lemma 1

PROOF. Since r_i and r_j are contradicting, the maximum satisfiable matching M_{ij} in G_{ij} must not simultaneously contain edges inducing r_i and edges inducing r_j. In other words, M_{ij} must either be a maximum satisfiable matching in G_i or in G_j, so that either $|M_{ij}| = |M_i|$ or $|M_{ij}| = |M_j|$. Since $|M_{ij}|$ is maximized, it follows that $|M_{ij}| = \max(|M_i|, |M_j|)$. □

C.3 Proof of Lemma 2

PROOF. Since $R_i \subseteq R_j$, the edges of M_i are all included in the constrained bipartite graph G_j constructed from R_j, so that M_i is a satisfiable matching in G_j. On the other hand, M_j is a *maximum* satisfiable matching in G_j, so we must have $|M_i| \leq |M_j|$. □

C.4 Proof of Algorithm SI correctness

LEMMA 3. *Given a CBG G, algorithm SI returns a maximum satisfiable matching of G correctly.*

PROOF. If no pruning occurs, all relationship subsets with no contradicting total-order relationships will be examined as in the algorithm SI, so that the matching reported in the end (which is the one whose size is largest among all maximum matchings) must be a maximum satisfiable matching of G. □

C.5 Proof sketch of the optimal integration plan problem

Running time breakdown of all SIs executed in part (B)	SF=1.0	SF=5.0	SF=10
T1 Constructing flow networks	670s	3062s	5864s
T2 Maximum flow algorithm	0.064s	0.035s	0.092s
T3 Merging tuples & inserting them into SDBs	0.047	0.049s	0.045s
T4 Algorithmic work	14870s	14887s	14867s
$\sum_{SI}$	15541s	17949s	20732s

Effectiveness of SI algorithm			
E1. Total num. of nodes in all SDBs	918K	4.1M	8.2M
E2. Total num. of nodes in all flow networks	30	32	32
	—the below is same for all scales—		
Table	customer	orders	district
Pruning Effectiveness	E3 / E4 / E5	E3 / E4 / E5	E3 / E4 / E5
$SI(D_1, D_2) \rightarrow D_a$	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
$SI(D_a, D_{12}) \rightarrow D_b$	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
$SI(D_b, D_4) \rightarrow D_c$	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
$SI(D_c, D_6) \rightarrow D_d$	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
$SI(D_d, D_{13}) \rightarrow D_e$	2 / 1 / 1	0 / 0 / 0	0 / 0 / 0
$SI(D_e, D_{11}) \rightarrow D_f$	0 / 0 / 0	64 / 1 / 31	0 / 0 / 0
$SI(D_f, D_{14}) \rightarrow D_g$	512 / 1 / 255	0 / 0 / 0	4 / 1 / 1
$SI(D_3, D_{10}) \rightarrow D_h$	512 / 4 / 266	0 / 0 / 0	0 / 0 / 0
$SI(D_h, D_5) \rightarrow D_i$	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
$SI(D_i, D_{15}) \rightarrow D_j$	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
$SI(D_j, D_7) \rightarrow D_k$	16777216 / 2 / 8388607	0 / 0 / 0	0 / 0 / 0
$SI(D_k, D_9) \rightarrow D_l$	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
$SI(D_l, D_{16}) \rightarrow D_m$	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
$SI(D_m, D_8) \rightarrow D_n$	0 / 0 / 0	0 / 0 / 0	64 / 2 / 34

Figure 15: Details of SI algorithm (TPC-C)

PROOF SKETCH. Given an instance of a cross product optimization [7], we create a corresponding symbolic database such that the matching size between two databases is always equal to the size of the cartesian product of the databases plus the size of the two databases. Thus, finding the maximum (satisfiable) matching equals to finding the optimal join ordering, which is $\mathcal{NP}$-hard. □

D. SUPPLEMENTARY EXPERIMENTAL INFORMATION

D.1 Experimental Setup

QAGen uses PostgreSQL to manage the symbolic/instantiated database and uses Java to implement the SQP operations. For easy interacting with QAGen's components, we also use Java and PostgreSQL to implement MyBenchmark. All experiments were carried out on a Pentium Dual-Core 2.5GHz PC with 8GB memory running Ubuntu. In all experiments, we set the relative error tolerance to be 100% for cardinalities in range [1, 1000] (e.g., the acceptable range of cardinality 10 is [1, 20]; cardinality 0 is excluded), 10% for cardinalities in range [1001, 10000] (e.g., the acceptable range of cardinality 5000 is [4500, 5500]), and 1% for cardinalities >10001.

D.2 Experimental Result: TPC-C Benchmark

The TPC-C benchmark models a typical OLTP environment where users executes transactions against a database. We downloaded an open-source implementation of TPC-C from http://db.apache.org/derby/index.html. We refer readers to the TPC-C specification for the details of the queries (we name the TPC-C queries according to their appearance order in the specification). After removing IUD and independent queries, 16 TPC-C queries remained. The expected cardinalities annotated on the operators of the queries are specified according to the actual cardinalities obtained by running the queries on the TPC-C data with scale factor 1.0 (they are scaled-down to a scale factor of 0.1 during the plan search process). The breakdown of the whole plan search process is as follows:

Item	Description / Sub-item	Time
$\mathcal{A}_1$	scale-down the input queries	<1s
$\mathcal{A}_2$	SQP all down-scaled queries once [7]	50min
$\mathcal{A}_3$	Build a summary (graph) of MSM size	403s
$\mathcal{A}_4$	Suggest a plan P by finding a MST from the graph	0.5s
	$\sum_{\mathcal{A}}$	57min

Again, the most time consuming part is running SQP on all scaled-down input queries to generate the small SDBs ($\mathcal{A}_2$). The time spent on running SI on C_2^{16} pairs of SDBs ($\mathcal{A}_3$) is 403s.

The following table shows the experimental result about step $\mathcal{B}$, i.e., executing the good integration plan in scale 1.0:

Item	Description/Sub-item	Total Time Spent
$\mathcal{B}_1$	SQP all queries in original scale [7]	3hr55min
$\mathcal{B}_2$	Follow P to run SI operations	4hr10min
	(a) After each SI, instantiate the resulting SDB	2hr43min
	(b) Pose all the queries on the resulting database to check quality	0.52s
	$\sum_{\mathcal{B}}$	10hr56min

The overall running time, $\sum_{\mathcal{A}} + \sum_{\mathcal{B}}$, is 11 hours 53 minutes. By following the suggested plan, two databases D_g and D_n were generated. When posing the original queries on the generated database, all queries obtain *exact* cardinalities as annotated in the input.

Figure 15 presents the time breakdown of all executed SI operations in different scale factors (1.0, 5.0, and 10.0), using the best plan found in part $\mathcal{A}$. Most items behave the same as TPC-W workload. We can see that the suggested plan favors the integration of SDBs without any common table (as their MSMs are set to be the largest possible integer in those cases), so the first few SI operations do not integrate anything. Slightly different from the TPC-W experiment, the time spent on the SI algorithm (T4) dominates the overall running time. When we look at the number of cases (E3), we quickly find out that is related to the current implementation (not algorithmic issue) of MyBenchmark. Specifically, this TPC-C workload has one SI operation that needs to deal with 24 distinct relationship, leading to 16+ million cases. That should not be an issue originally because only two CBGs (E4) were actually processed after a large number of cases were pruned (E5). However, the current MyBenchmark implementation is implemented in

Num. of Queries	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\sum_{\mathcal{A}}$ (in seconds)	192	380	574	773	974	1179	1388	1600	1815	2033	2255	2481	2710	2942	3177	3415
$\sum_{\mathcal{B}}$ (in seconds)	1497	2380	3878	5375	6873	8370	9867	11365	12862	14362	15860	17358	33815	20353	21851	38996
Num. of DB	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2

Figure 16: Increasing the number of TPC-C queries

Java and uses an external constraint solver called Cogent [8], which is a C++ binary executable, to check Lemma 1. For each cross-language Cogent call, it took about 0.3s overhead (by JNI). All together there were 49632 Cogent calls (same for all scales; because that depends on the number of distinct relationships, not the data size). So, those calls used a total of 14850 seconds, which almost equals to time T4. Indeed we tried to use some constraint solvers written in Java in our implementation and that bottleneck was gone (the bottleneck is back to T1). However, we found that, in general, Java constraint solvers are not very stable. As an experimental prototype, we keep Cogent in our current implementation because it is more stable (most constraint solvers are written in C++). We are currently testing a more stable Java constraint solver that can replace Cogent, and we are considering to re-implement the whole SQP and MyBenchmark in C++ (so that it can work seamlessly with Cogent). As a side-note, the above also explains why item $\mathcal{B}_2$ becomes the bottleneck in part $\mathcal{B}$. That is also due to the large overhead spent on calling a non-Java external binary. Therefore, if we find a stable Java constraint solver, the running time of $\mathcal{B}$ can be reduced by 4 hours.

Other than the above implementation issue, this experiment draws similar conclusions as in the TPC-W experiments. The running time scales linearly to the workload size. The experimental result about varying the number of annotated queries in the input is summarized in Figure 16. Overall, the running time scales roughly linearly to the number of input queries. All the generated databases are perfect (i.e., no error). When the 9-th query was added, one more database was required.

Acknowledgment. We thank Byron Choi, Man Lung Yiu, Ming-Hay Luk, Duncan Yung, and the anonymous reviewers for their insightful comments.