ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/354065395

MyBenchmark: generating databases for query workloads

Conference Paper · March 2014

CITATIONS
0

READS
14

1 author:

Wilfred W. K. Lin
HerbMiners Informatics Limited
169 PUBLICATIONS **194** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Deep Learning on Knowledge Graph and Mind Map View project

MetaVerse View project

All content following this page was uploaded by Wilfred W. K. Lin on 23 August 2021.

The user has requested enhancement of the downloaded file.

The VLDB Journal
DOI 10.1007/s00778-014-0354-1

REGULAR PAPER

MyBenchmark: generating databases for query workloads

Eric Lo · Nick Cheng · Wilfred W. K. Lin · Wing-Kai Hon · Byron Choi

Received: 2 July 2013 / Revised: 10 January 2014 / Accepted: 13 February 2014
© Springer-Verlag Berlin Heidelberg 2014

Abstract To evaluate the performance of database applications and database management systems (DBMSs), we usually execute workloads of queries on generated databases of different sizes and then benchmark various measures such as respond time and throughput. This paper introduces MyBenchmark, a parallel data generation tool that takes a set of queries as input and generates database instances. Users of MyBenchmark can control the characteristics of the generated data as well as the characteristics of the resulting workload. Applications of MyBenchmark include DBMS testing, database application testing, and application-driven benchmarking. In this paper, we present the architecture and the implementation algorithms of MyBenchmark. Experimental results show that MyBenchmark is able to generate workload-aware databases for a variety of workloads including query workloads extracted from TPC-C, TPC-E, TPC-H, and TPC-W benchmarks.

Keywords Query Processing · Performance · Benchmarking · Data Generation

Research supported by Grant PolyU 525009E from Hong Kong RGC.

E. Lo (✉) · N. Cheng · W. W. K. Lin
Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Hong Kong
e-mail: ericlo@comp.polyu.edu.hk

W.-K. Hon
Department of Computer Science, National TsingHua University, Hsinchu City, Taiwan
e-mail: wkhon@cs.nthu.edu.tw

B. Choi
Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, Hong Kong
e-mail: choi@hkbu.edu.hk

1 Introduction

Query performance is a key factor of successful database applications and database management systems (DBMS). To evaluate their performance, we usually execute workloads of queries on generated databases of different sizes and then benchmark various measures such as respond time and throughput.

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 of different sizes with an additional ability 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:

1. **DBMS testing** Recent papers [7,9,22,25] have pointed out that controlling the cardinalities of query operators in a test query is very useful in DBMS testing. For example, test engineers can study the performance of a hash-join implementation by varying the input and output cardinalities of the join operator [9]. Recent data generation technology has made some progress in this respect. For example, QAGen [7] 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 one test case as input and generates an independent test database that is specific for

Published online: 15 March 2014

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.

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]

2. **Stress testing database applications** Consider a database 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 1 GB 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]
3. **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 database applications. By using MyBenchmark, an enterprise is able to study the performance of a DBMS with respect to its own database applications. Suppose a new start-up is choosing a DBMS product. The start-up may wish to know which DBMS (e.g., Oracle, Microsoft 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 when making the final decision.

A preliminary version of this paper appeared in [23], which supports only SELECT-PROJECT-JOIN (SPJ) queries. This new version of MyBenchmark has the following new features: (i) It supports SPJ queries with GROUP-BY and HAVING clause and (ii) parallel data generation. Specifically, the new technical contents over the previous version of MyBenchmark [23] include:

1. [Extended Core Algorithm] The core algorithm used in the previous version of MyBenchmark was an exact algorithm [23]. In this version, we have added a new randomization phase and a new approximation phase to the core algorithm. Furthermore, the compression method in the core algorithm has been extended in order to support the GROUP-BY and HAVING clause. These extensions are necessary when dealing with complex workloads like TPC-H. (Sect. 3.2.1)
2. [Bushy Integration Plans] The main methodology of MyBenchmark is to integrate multiple query-aware test databases into one workload-aware test database. There are a variety of orderings to do so, which is similar to the traditional join ordering problem. In the previous version of MyBenchmark [23], only left-deep integration plans were considered. In this paper, we prove that bushy plans can yield higher quality generated databases. Therefore, this new version of MyBenchmark considers both left-deep and bushy integration plans. (Sect. 3.3)
3. [Top–down Data Instantiation] The new version MyBenchmark uses a top–down approach to instantiate data values, instead of a bottom–up approach as in the previous version [23]. Such architectural change significantly speeds up one of the most time dominating steps of MyBenchmark. (Sect. 4)
4. [Parallel Implementation] In order to support parallel data generation, a few related components in MyBenchmark are re-implemented.
5. [New Experiments and Better Results] Experiments in [23] only covered TPC-C and TPC-W workloads. In this version, experiments now also cover TPC-E and TPC-H workloads. Experimental results show that the new version of MyBenchmark can now return a single database for TPC-C and TPC-W workloads (the version in [23] had to return two databases for each workload). New experimental results show that the new version of MyBenchmark can return a single database for TPC-E and TPC-H workloads. Furthermore, the results show that the new version of MyBenchmark can generate database much more efficiently due to the new changes in the core algorithm, use of parallel machines, and top–down data instantiation. (Sect. 5)

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 presents the experimental results. Section 6 concludes the paper. The "Appendix" contains some supplementary details.

[1] In case MyBenchmark generates more than one test database, we may use a database testing framework (e.g., DbUnit [1], HTpar [17]) 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.

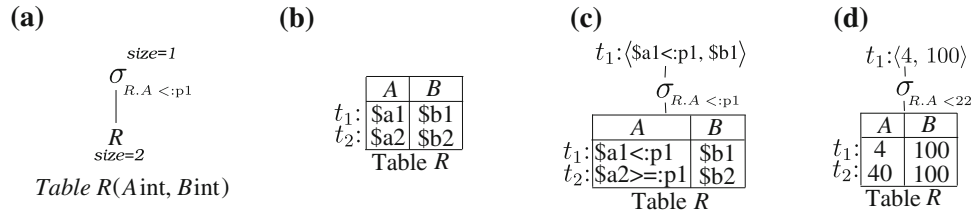

Fig. 1 Symbolic query processing (SQP), **a** annotated query Q_1 (:p1 is a parameter), **b** the initial symbolic database for Q_1, **c** D_1; Q_1-aware symbolic database (SDB), **d** Q_1-aware database D_1

2 Background and related work

There are a number of commercial and academic data generators (e.g., [2,3,8,10,16,19,20,26,30,31]) that generate synthetic databases for a given database schema. Beside accepting the database schema as input, some tools also accept table sizes, data repositories, and additional data constraints (e.g., statistical distributions of individual attributes and value ranges) as inputs. Recently, the big data era has motivated the research of generating data in larger volume by using multiple machines (e.g., [28]) and with a higher variety (e.g., [15,18]). However, all these tools do not generate data that can control the query characteristics. Unless tuning the generated data manually, queries extracted from real database applications easily obtain empty results from those data. As a result, the generated data may not be useful to our applications such as DBMS testing.

Query-aware data generation was first studied by [24] and has received renewed attention in recent years. In [24], Mannila and Räihä studied the generation of test data that complies with functional dependencies for simple relational queries. In [6], Binnig et al. studied the generation of test data for functional testing database applications. The focus of [6] is to generate *minimal size* test databases for a *single* application query. In [27], Olston et al. 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. In [5], Arasu et al. first transformed a workload of queries into a linear program. Then, the solution of the linear program is represented as a probability distribution of the data to be generated and data records are generated from the distribution. This approach, however, cannot handle a rich set of SQL queries (e.g., queries with the HAVING clause) and cannot easily support parallel data generation.

We now give a brief background on QAGen. QAGen 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). Figure 1d shows the output of QAGen for Q_1, which is a query-aware database D_1 that conforms to H, and a set of parameter values P. Executing query Q_1 (with parameter `:p1=22`) on D_1 guarantees that the constraints defined on Q_1 are satisfied.

To process a query like the one in Fig. 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 (Fig. 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, the operator 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 (Fig. 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 (Fig. 1d). The SQP engine in QAGen includes the SQP implementations of most standard SQL operators. QAGen is thus able to generate databases for a variety of SQL workloads.

SQP is an advanced data generation technology that is much more complicated than traditional query-unaware data generation technology such as [16]. Therefore, query-aware data generation tools usually have a longer running time and run as an offline (background) process [22]. Nevertheless, the advantage of SQP is that it naturally supports more SQL operators and can be easily parallelized using n machines to SQP n queries in parallel.

3 MyBenchmark

MyBenchmark uses the symbolic query processing technique developed in QAGen 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 $\mathcal{Q} = \{\mathcal{Q}_\infty, \mathcal{Q}_\in, \ldots, \mathcal{Q}_\setminus\}$ (the operator(s) in $\mathcal{Q}_i$ are annotated with cardinality constraint(s) C_i defined over its subexpression, e.g., see Fig. 2), 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

Sub-expression	Card.
1. SELECT * FROM R	2
2. SELECT * FROM R WHERE R.A < :p1	1

Fig. 2 Formal specification of a query

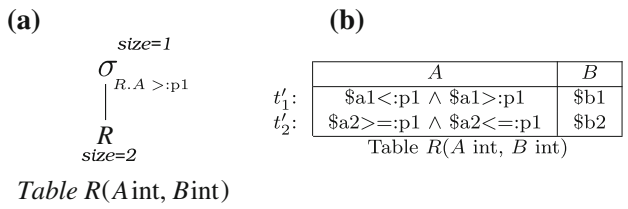

Fig. 3 Examples of SQP on a "processed" SDB. **a** Annotated query Q_2 (:p1 is a parameter), **b** an unsatisfiable symbolic database (due to t_1) after bad SQP Q_2 on Fig. 1c's D_1

$(1 \leq j \leq m)$ 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 Sect. 3.2). Approximate cardinalities are sufficient for applications such as DBMS testing [9,25] and database application testing [6]. Assume that a DBMS test engineer wants to use MyBenchmark to generate a workload with a 1 GB 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 acceptable. As SQP controls the data distributions through the operator cardinalities [7], we focus on the control of the operator cardinalities.

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. For applications like stress testing database applications and application-driven benchmarking, it would be advantageous to carry out testing or benchmarking based on a single (or fewest) 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 Fig. 1a, and Q_2 be a selection query in Fig. 3a, which specifies that table R should have two tuples and the query should return one tuple.[3] Assume Q_1 is first symbolically processed by the SQP engine and we obtain the symbolic database D_1 (Fig. 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 SDB (Fig. 3b) in which tuple t'_1 is associated with a contradicting constraint [$a1<:p1 ∧ $a1>:p1].

Figure 4 shows the architecture of 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 Fig. 3b) 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 Integration Planner is designed for integrating multiple SDBs and we defer its discussion until Sect. 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 at the end of this subsection and the integration of multiple symbolic databases in Sect. 3.3.

We use the annotated SQL queries Q_3 and Q_4 in Fig. 5a, c as the running example. For ease of exposition, both Q_3 and Q_4 are simple selection queries posed on table S. Figures 5b, d 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

[3] 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.

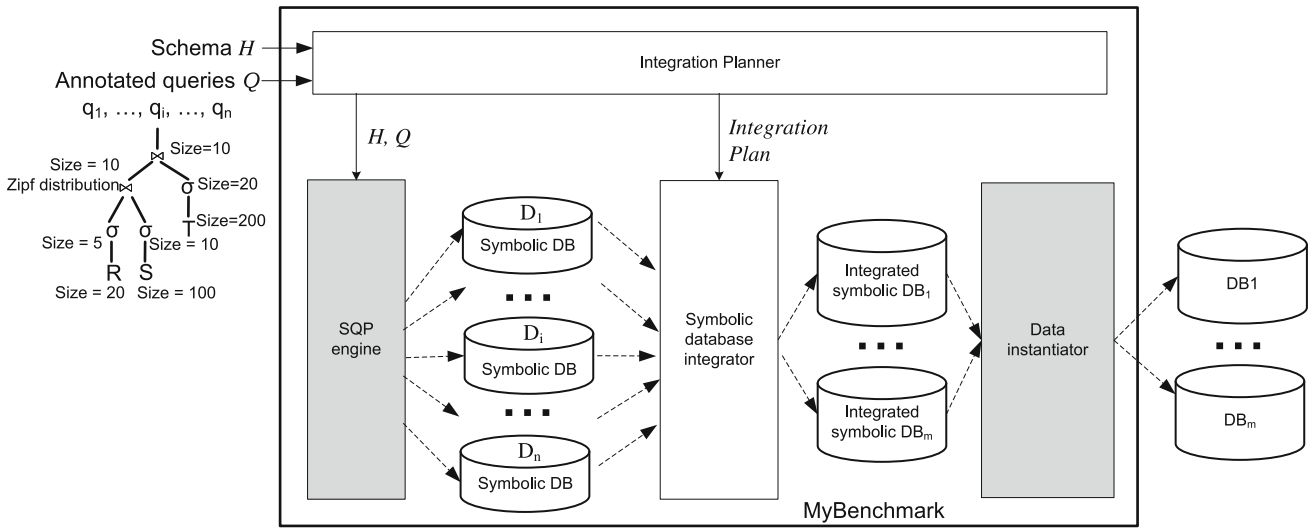

Fig. 4 MyBenchmark architecture

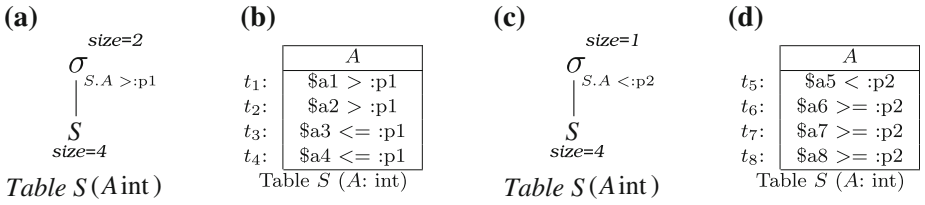

Fig. 5 Examples for symbolic database integration. **a** Annotated query Q_3 (:p1 is a parameter), **b** D_3, **c** annotated query Q_4 (:p2 is a parameter) **d** D_4

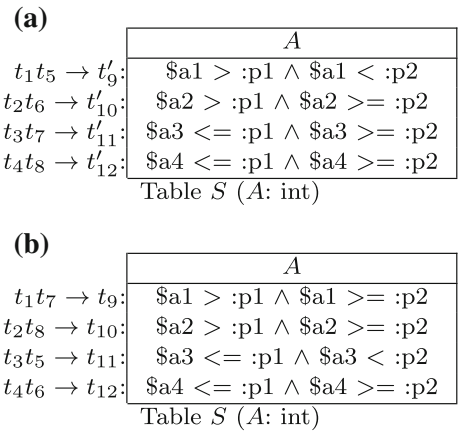

Fig. 6 SDBs integrated from D_3 (Fig. 5b) and D_4 (Fig. 5d), **a** $\bar{D}$ (t'_9 contradicts t'_{11} and t'_{12}), **b** ideal integrated symbolic database

in an integrated symbolic database $\bar{D}$ as shown in Fig. 6a. 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 ≤ :p1. As such, the integration algorithms should be designed to minimize the number of symbolic tuples with contradicting constraints in the integrated SDB. For example, Fig. 6b 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 formulate a graph problem.

Definition 1 (CONSTRAINED NODE). A node n is *constrained* if 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* if 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), where 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.

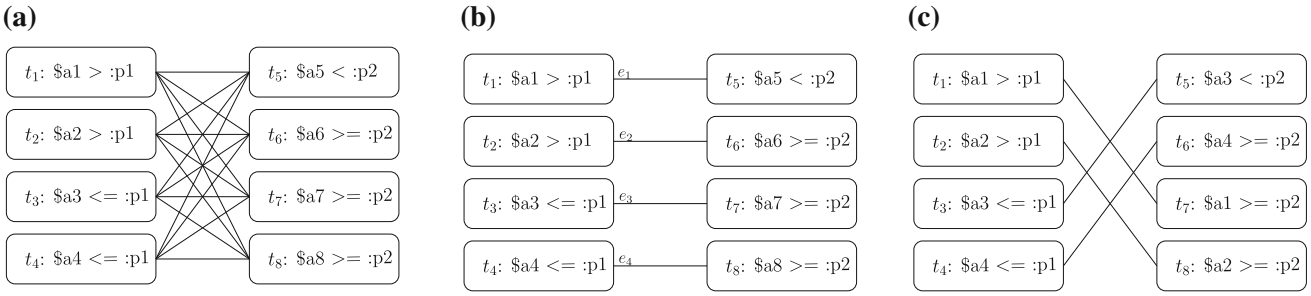

Fig. 7 **a** A constrained bipartite graph G_a modeling the integration of databases D_3 and D_4 in Fig. 5, **b** a maximum but not maximum satisfiable matching M_1, **c** a maximum satisfiable matching M_2

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 Fig. 5b, d can be modeled as a constrained bipartite graph G_a shown in Fig. 7a.[4] 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.

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 7b 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 Fig. 5b should be integrated with tuple t_5 of D_4 shown in Fig. 5d. Therefore, if the integration follows M_1, the resulting integrated database would become $\bar{D}$ in Fig. 6a. On the other hand, if the integration follows M_2 (see Fig. 7c), which is a maximum satisfiable matching of G_1, the resulting integrated database would become the ideal integrated symbolic databases shown in Fig. 6b.

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 7.1"). The main difficulty lies in the requirement of "satisfiability" among the induced relationships of variables at run-time (e.g., in Fig. 7b, 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 Sect. 3.1) is not a good idea.

In MyBenchmark, we have developed many techniques to avoid resolve that problem. Specifically, we have developed a best-effort symbolic database integration algorithm that utilizes the special properties of SQP to reduce the search space of symbolic database integration. Our experiments show that our symbolic database integration algorithm can find a high-quality MSM quickly and scale well under a variety of inputs.

3.2.1 The symbolic database integration algorithm

The symbolic database integration algorithm (*SI*) we developed for MyBenchmark solves the maximum satisfiable matching (MSM) problem 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.

Example 1 Assume that *SI* takes G (Fig. 7a) as input. As a first step, the total-order relationship r_1=[:p2 > :p1] induced by edges (t_1, t_5) and (t_2, t_5) and the 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 identified and 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 Fig. 10a) includes the edges that induce the total

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

order r_1 and the edges that induce no total order (e.g., (t_1, t_6)). G_2 (shown in Fig. 10b) 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 in Fig. 7a). G_4 includes the edges that induce no relationships. (We do not present G_4 here for brevity.)

By following the basic idea illustrated above, *SI* has to find the maximum matching for each of the $2^{|\mathcal{R}|}$ constrained bipartite graphs. In practice, $|\mathcal{R}|$ is actually a fairly small number in many real workloads because $|\mathcal{R}|$ increases only when two queries have parametric predicates that access the same attribute of the same table. For example, in our experiments, the maximum values of $|\mathcal{R}|$ found in TPC-C, TPC-E, and TPC-W benchmark are 9, 8, and 13, respectively.[5] Furthermore, we have incorporated pruning techniques into *SI* such that it prunes most of the CBGs in practice. In the previous version of MyBenchmark [23] that supported only SPJ queries, our SI algorithm was an exact algorithm that examines all the enumerated CBGs that are not pruned by our pruning techniques. In this version, we extend the SI algorithm to be a randomized/approximation algorithm because we found that complex workloads like TPC-H has a $|\mathcal{R}|$ value of 28, in which the exact algorithm still has to examine too many CBGs even with the pruning techniques.[6]

In the following, we first present three techniques that prune CBGs in *SI*:

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 $|M_{ij}| = \max(|M_i|, |M_j|)$.*

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 $|M_{ij}| = \max(|M_i|, |M_j|)$. □

[5] Although TPC-E has more queries than TPC-C, their maximum $|\mathcal{R}|$ values are close because TPC-E also has more tables and the parametric predicates often access different attributes of different tables.

[6] TPC-H has a much higher $|R|$ value than the other workloads because TPC-H queries often have parametric predicates on the same attributes of the fact table (e.g., the `l_shipdate` attribute in `lineitem` table).

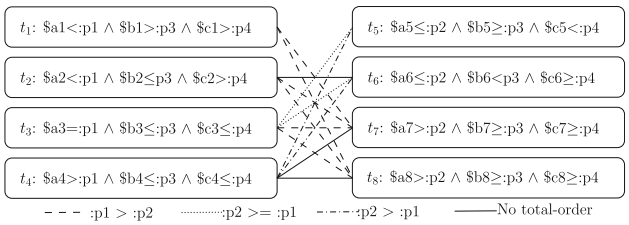

Fig. 8 A CBG for a multiple query integration instance

Example 2 Following up Example 1, by Lemma 1, *SI* does not need to examine G_3 (7a) 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 (Fig. 10a) and the MSM of G_2 (Fig. 10b).

2. Subset pruning Consider Fig. 8, 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 happens when multiple queries are input to the system (see Sect. 3.3 for details). For the time being, we focus on an MSM search for Fig. 8.

In Fig. 8, the set of edges induces the following set of total-order relationships $\mathcal{R} = \{r_1 = [:p1 > :p2], r_2 = [:p2 \geq :p1], r_3 = [:p2 > :p1]\}$. 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 are no total orders induced from symbols connecting parameters :p3 and :p4. For the example in Fig. 8, we can visualize its 2^3 cases (all possible subsets) as a search tree (Fig. 9). 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 Fig. 9, 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).

Next, we introduce another lemma that can further prune the search space:

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 size of the MSM M_j, of the constrained bipartite graph constructed from R_j (i.e., $|M_i| \leq |M_j|$).*

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 $|M_i| \leq |M_j|$. □

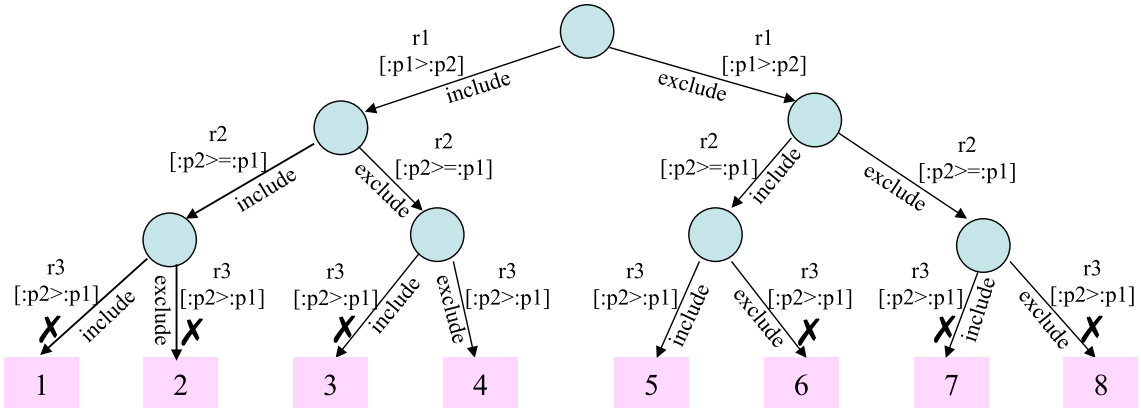

Fig. 9 Search tree of SI. Lemma 1 prunes cases 1, 2, and 3. Lemma 2 prunes cases 6, 7, and 8

By Lemma 2, SI can prune cases 6, 7, and 8 in Fig. 9 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.

3. Early stopping Our goal is to find the largest MSM among all the possible CBGs. So, whenever SI finds a perfectly satisfiable matching in a CBG, it can stop early. Although simple, experimental results (Table 2) show that this technique is very useful since SI is often able to find a perfectly satisfiable matching very early in the process.

Our experiments show that the three pruning techniques above are very effectiveness and they together could skip many cases in the experiments. Therefore, our previous version of MyBenchmark [23] is based on those to implement an exact SI algorithm, where we found that was sufficient to handle TPC-C ($|\mathcal{R}| = 9$) and TPC-W workloads ($|\mathcal{R}| = 13$) [23]. In this version, when we extend MyBenchmark to deal with workloads with TPC-H workload, we find that $|\mathcal{R}|$ is significantly large ($|\mathcal{R}| = 28$) that renders the original exact algorithm run too slow even with pruning techniques 1, 2, and 3. Therefore, we further extend the algorithm to examine only a limited number of cases among of all unpruned cases. The detailed is as follows.

4. Randomization and approximation In this new version of MyBenchmark, we allow the user to specify her expectation Pr of getting the best-$T\%$ solution. Then, the SI algorithm randomly examines only s unpruned cases (it still early stops if it finds a perfect satisfiable matching early). The value s is computed by:

$$1 - (1 - T\%)^s \geq Pr \tag{1}$$

Example 3 Suppose a user wants to ensure that each SI operation returns a satisfiable matching whose size is better than 99 % of other unpruned cases (i.e., $T = 1\,\%$) with a probability Pr of 0.99. Then, SI can achieve that by examining only $s = 459$ cases. In fact, that is already too many for TPC-C, TPC-E, and TPC-W in our experiments where SI always finds a perfectly satisfiable matching and stops early before 459 cases are examined.

In addition to the use randomization to improve the efficiency of SI, in this version, we also add the following *approximation phase* to SI in order to bound its approximation ratio:

a. Find any maximum matching M in G, the given CBG that models the integration of an SDB D_i and D_j
b. Let $p_1, p_2, \ldots, p_k$ be the k parameters involved in the induced total-order relationships.
c. Select a k-permutation $\sigma : [1, k] \rightarrow [1, k]$ uniformly at random.
d. Order the parameters by $p_{\sigma(1)} < p_{\sigma(2)} < \ldots < p_{\sigma(k)}$.
e. Output all edges in M whose induced relationships do not violate the above ordering.

Lemma 3 *The above approximation phase in SI returns a satisfiable matching, whose size is at least half of an MSM.*

Proof It is trivial that the edges returned form a satisfiable matching. For each edge in M, the probability that it is output is at least $1/2$, so that the expected size of the returned matching is $|M|/2$. By de-randomizing the approximation phase using standard method of conditional expectations, we know that the above approximation phase runs in polynomial time and reports a satisfiable matching whose size is at least half of an MSM. □

In the following, we present the last trick to further optimize the efficiency of SI by compressing the problem instances.

5. Compressing the problem instances SI's efficiency can be further improved by *compressing* the symbolic tuples. For instance, in Fig. 10a, tuples t_1 and t_2 are capturing the same selection predicate `S.A > :p1` of

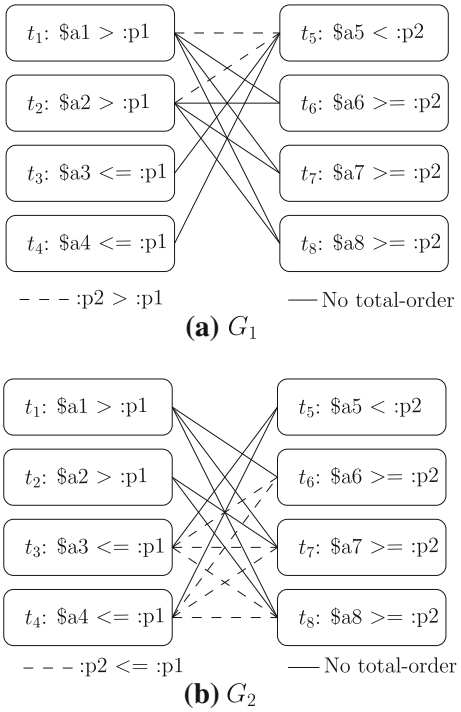

Fig. 10 Two constrained bipartite graphs

query Q_3. Therefore, they are compressed into a *single* node. A maximum matching problem is often transformed into a maximum flow problem in network optimization [4,21]. 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. (Connect 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 $min(|N_u|, |N_v|)$.

Figure 11a, b show the flow networks compressed from Fig. 10a, b, respectively. We can see that the number of nodes is only half of the original constrained bipartite graph. The compression scheme described above covers cases where the inputs are simple queries without grouping and joining (SQP performs symbol replacements in those cases), which is covered by our previous version of MyBenchmark [23]. In the following, we present new material about the compression scheme when the queries involving grouping and joining.

In SQP, two identical symbols shall be instantiated by the same value. So, in SQP, the grouping operator and the joining operator are implemented to replace symbols. For example, Fig. 12b shows a symbolic relation after SQP, where the query contains a selection >:p1 which expects 6 tuples pass the filter and those tuples are grouped into two groups. The original symbolic relation before SQP is shown in Fig. 12a. To control that only two groups are returned, the grouping operator *replaces* symbols $a2, $a3 by symbol $a1, symbols $a5, $a6 by symbol $a4. (Other replacement scheme is doable, and that depends on how the user wants the value distribution looks like; see [7] for details).

With symbol replacement, compressing tuples shall not only base on the predicate, but also base on the symbol names. For example, tuples t_1, t_2, and t_3 shall be compressed into one node, and tuples t_4, t_5, and t_6 shall be compressed into another node. In Fig. 12b, after compression, we obtain three compressed nodes with edge weights 3, 3, and 2, respectively, as shown in Fig. 12c. We name nodes obtained by compressing

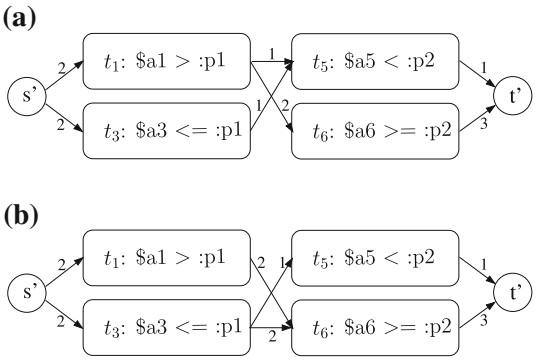

Fig. 11 Flow networks compressed from constrained bipartite graphs in Fig. 10. **a** G'_1, **b** G'_2

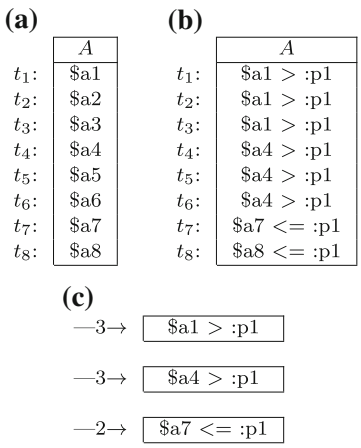

Fig. 12 Symbol replacement in SQP. **a** Before SQP, **b** after SQP, **c** nodes in flow network

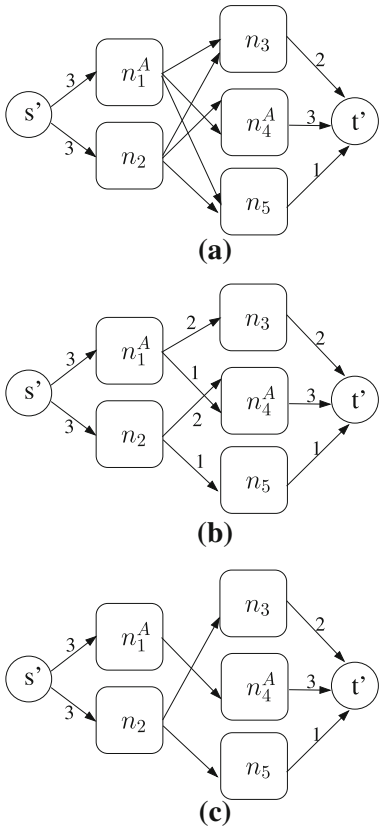

Fig. 13 Flow networks with compressed nodes

duplicate symbols as *atomic* nodes and name the others as *normal* nodes.

Figure 13a shows a flow network which involves atomic compressed nodes because of the existence of group-by or join operators (we mark atomic nodes with a superscript A). Without distinguishing atomic nodes from normal nodes, the matching quality may be affected. For example, Fig. 13b shows a possible maximum flow of Fig. 13a. The maximum flow suggests that the three tuples in the atomic node n_1^A should be matched to two tuples in node n_3 and one tuple in node n_4^A. So, tuples that are originated from the same group, such as tuples t_1, t_2, and t_3 in Fig. 12b, will then integrate with two sets of tuples, resulting them into two separate groups (e.g., t_1 and t_2 in one group, and t_3 is another group) in the integrated database.

To resolve this issue, this new version of MyBenchmark takes special care when constructing the flow network if atomic nodes exist. Specifically, we connect nodes with satisfiable edges in three rounds. In the first round, we connect an atomic node from U to an atomic node from V only if they have the same in(out)-edge weight. If such a pair exists, they will not connect to the other nodes anymore. In the second round, if there is an atomic node from U remains unconnected, then we connect it to a normal node from V that shares the same in(out)-edge weight. Again, if such a pair exists, they will not connect to the other nodes anymore. In the last round, we connect the remaining nodes together, based on the original flow network construction requirements. By following this principle, the flow network in Fig. 13a becomes the flow network in Fig. 13c. We can see that n_1^A and n_4^A are connected to each other and no other nodes connect to them. That ensures that all three tuples (of the same group) from node n_1^A merge with all three tuples (of the same group) from node n_4^A.

Algorithm *SI*—the pseudo-code Algorithm 1 summarizes the discussion above and presents the pseudo-code of *SI*. Briefly, steps (1) to (2) implement Lemmas 1 and 2 to prune CBGs. Step 4 describes the examination of a sample s unpruned CBGs, the compression of problem instances, as well as the early stopping condition. 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 $\mathcal{O}(n^3)$ [12] (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. Step 5 is the approximation phase that bounds the approximation ratio of the algorithm. The maximum matching algorithm used in that step is the same the push-relabel maximum flow algorithm used in Step 4(c). Finally, in Step 7, *SI* follows the largest maximum satisfiable matching that it has found to perform tuple integration.

3.2.2 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, Fig. 8 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 common 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 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.

Algorithm 1 $SI(D_i, D_j)$

(0) Scan SDB D_i and SDB D_j to construct the CBG.
(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=$_{\text{null}}$ to store the largest MSM discovered so far.
(4) Randomly pick an unpruned case in T if the total number of cases examined is less than or equal to s cases
  (a) construct a new constrained bipartite graph G_i which includes
    (i) the edges that induce the relationships in R_i 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) Find a basic approximation solution for SI:
  (a) find any maximum matching M in G
  (b) let $p_1, p_2, ..., p_k$ be the k parameters involved in the induced total-order relationships.
  (c) select a k-permutation $\sigma : [1, k] \rightarrow [1, k]$ uniformly at random
  (d) order the parameters by $p_{\sigma(1)} < p_{\sigma(2)} < \ldots < p_{\sigma(k)}$
  (e) set BASIC- MSM = all edges in M whose induced relationships do not violate the above ordering.
(6) Set MAX- MSM = BASIC- MSM if size-of BASIC- MSM > size-of MAX- MSM
(7) Follow MAX- MSM to perform the integration.

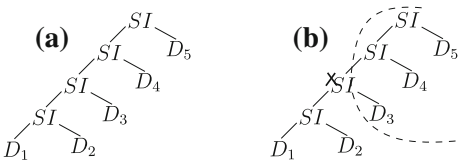

Fig. 14 **a** An integration plan, **b** an execution plan

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 14a shows a left-deep 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 has the early stop strategy that returns 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 creates a new database 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 Fig. 14a 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 Fig. 14b). 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 [13], which is $\mathcal{NP}$-hard (see "Appendix 7.2"). Traditional query optimization uses heuristics and cardinality 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 effi-

cient algorithm. For MyBenchmark, we pre-build a summary about the quality of some core SI operations and exploit that to estimate the best plan. 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 cardinalities requirements of the input queries (e.g., from generating 1 GB data to 1 K data) using the test case generation tool in [22]. For example, the input query in Fig. 5a 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 (2) the ratio between MSM size and the CBG size depend on the characteristics of the input queries (e.g., the selection predicates). In other words, (1) and (2) do not depend on the size of the databases to be generated. Therefore, there would be no difference between (1) and (2), and consequently the number of resulting databases, between generating 1 K and 1 GB data. However, there would be a significant time-cost difference between the two. More specifically, the running time of an SI operation mainly attributes to:

i. Scanning SDBs D_i and D_j to construct the CBG.
ii. For each unpruned case, computing the satisfiable matching on the compressed flow network.
iii. Merging tuples in D_i and D_j according to the MSM and inserting them into a new SDB D_{ij}.

(i) and (iii) incur significant time because QAGen (and thus MyBenchmark) store the symbolic/instantiated tuples in a RDBMS (*e.g.*, PostgreSQL) (the data are usually too large to fit into main memory). Therefore, much time is spent on the overhead of reading and writing symbolic tuples from and to the disk. 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 15a shows such a graph for our example. Recall that, one additional database is required whenever the quality of the resulting instantiated database drops below the user-defined

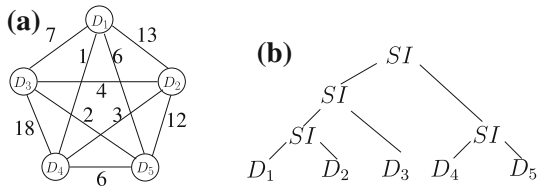

Fig. 15 **a** A graph representing the (scaled-down) MSM sizes between pairs of input SDBs, **b** a bushy integration plan deduced by recursively cutting the graph at minimum cut

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. In our previous version [23], we adopted common heuristics in query optimization and considered only left-deep plans. The integration plans were then derived from the maximum spanning tree (MST) of the graph. In this version, we also consider bushy plans in addition to left-deep plans because bushy plans allow better parallelism [11]. Different from traditional query optimization, considering bushy plans in addition to left-deep plans would not increase the plan search time (which is based on partitioning the summary graph at the minimum cut; details given later). In addition, bushy plans have the following advantage:

Lemma 4 *Among all different plans of integrating D_1, D_2, $\ldots$, D_n, bushy integration plans have a higher or equal chance of getting the MSM (i.e., the optimal solution) than left-deep integration plans.*

Proof Given a set of SDBs $\mathcal{D}$: $\{D_1, D_2, \ldots, D_n\}$. For analysis simplicity, assume the sizes of the SDBs are the same, denoted as $|D|$. For clarify, this proof argues with $n = 4$. We can generalize the argument to any value of n via a straightforward induction.

We use $J_{\mathcal{D}}$ and $B_{\mathcal{D}}$ to denote a left-deep integration tree and a bushy integration tree[7], respectively. Without loss of generality, the order of the leaf nodes of $J_{\mathcal{D}}$ and $B_{\mathcal{D}}$ are identical. We have $J_{\mathcal{D}} = SI(SI(SI(D_1, D_2), D_3), D_4)$ and $B_{\mathcal{D}} = SI(SI(D_1, D_2), SI(D_3, D_4))$.

Without prior knowledge to the optimum, we assume the number of satisfiable edges that induce relationship R_i is proportional to the number of clauses of the edges' propositional formulas. That is, denoted that the number of satisfiable edges of $\phi_1 \wedge \phi_2$ is $x\% \times |D|$ and the number of satisfiable edges of $\phi_1 \wedge \cdots \wedge \phi_n$ is $x\%^{n-1} \times |D|$.

Consider the integrations of $J_{\mathcal{D}}$. In $SI(D_1,D_2)$, SI determines the MSM from each relationship set R_i induced from $x\% \times |D|$ edges. Similarly, the number of satisfiable edges determined by $SI(SI(D_1,D_2),D_3)$ and $SI(SI(D_1,D_2),D_3)$, D_4) are $(x\% + x\%^2) \times |D|$ and $(x\% + x\%^2 + x\%^3) \times |D|$, respectively.

[7] A bushy tree is a balanced tree.

In comparison with $B_{\mathcal{D}}$, the edges of $SI(D_3, D_4)$ and $SI(SI(D_1,D_2), SI(D_3,D_4))$ are $x\% \times |D|$ and $x\%^3 \times |D|$, respectively. Therefore, when determining the maximum satisfiable matching, $B_{\mathcal{D}}$ always examined more satisfiable edges (*i.e.*, $(2x\% + x\%^3) \times |D|$) than $J_{\mathcal{D}}$ (*i.e.*, $(x\% + x\%^2 + x\%^3) \times |D|$).

Without prior knowledge to the optimum, and $B_{\mathcal{D}}$ and $J_{\mathcal{D}}$ determine MSMs from the same $\mathcal{R}$, $B_{\mathcal{D}}$ has a higher probability of getting the MSM than $J_{\mathcal{D}}$. □

Now, the remaining question is how to deduce an integration plan from a summary graph like Fig. 15a. Here, recursively partitioning the summary graph at the minimum cut becomes a natural choice. The recursion ends when the input graph contains less than or equal to two nodes. Figure 15b shows the deduced integration plan from the summary graph shown in Fig. 15a. $SI(D_1, D_2)$ and $SI(D_4, D_5)$, for example, can be executed in parallel.

The integration plan generation is taken care by the Integration Planner (see Fig. 4) of the system. Similar to traditional query optimization, our approach is also based on heuristics and estimation, which may not yield the optimal plan. Nevertheless, experimental results are encouraging. In our experiments on TPC-C, TPC-E, TPC-H, and TPC-W workloads, the integration plans returned by our new method successfully integrate all SDBs into one, whereas our previous integration method [23] that based on only left-deep plans could only integrate TPC-C and TPC-W workloads into two databases at best (we did not carry out experiments using TPC-E and TPC-H in [23]).

We end this section by discussing whether techniques like multi-query optimization [29] can be used in MyBenchmark in order to avoid generating SDBs for queries that share common subexpressions. Consider the following example, which consists of a workload of two queries:

Q_1 SELECT * FROM R JOIN S JOIN T;
Q_2 SELECT * FROM R JOIN S JOIN U;

In this case, multi-query optimization technique can detect the common subexpression of Q_1 and Q_2, that can help MyBenchmark to carry out symbolic query processing (SQP) on R and S only once but not twice (once for Q_1 and once for Q_2).

However, MyBenchmark often deals with workloads with parametric predicates and annotated with cardinality constraints like the following:

		Subexpr Card.
Q_3:	SELECT * FROM R JOIN S JOIN T	$\lvert\sigma_{a>:p1}R\rvert = 100$
	WHERE $R.a >: p1$ and $S.b <: p2$;	$\lvert\sigma_{b<:p2}S\rvert = 200$
Q_4:	SELECT * FROM R JOIN S JOIN U;	$\lvert\sigma_{a>:p3}R\rvert = 300$
	WHERE $R.a >: p3$ and $S.b <: p4$;	$\lvert\sigma_{b<:p4}S\rvert = 400$

In general cases like this, multi-query optimization techniques cannot help. In the example above, the SDB generated for table R in Q_3 should be different from that for Q_4 because (i) `:p1` does not necessarily equal to `:p3`; and (ii) to have only one symbolic relation R that satisfies the constraints imposed by both Q_3 and Q_4, we have to minimize the number of tuples that induce contradicting relationship in R—that is exactly the problem that algorithm SI is solving. In order words, the elimination of some SDBs by identifying common subexpressions in multiple queries can only be applied when (i) we are sure that `:p1 = :p3` **and** (ii) their respective subexpression cardinalities are the same. As multi-query optimization techniques could be used in only some very simple cases but not in general, we have not added this feature into MyBenchmark.

4 Summary of the methodology

Overall, the execution of MyBenchmark is composed of three phases:

Phase $\mathcal{A}$. Finding a good integration plan P. This involves:

($\mathcal{A}_1$) Scale down the cardinalities in the input queries. This is done by the tool in [22] 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 [7] for each input query.[8]

($\mathcal{A}_3$) Build an MSM size summary graph by running SI on every pair of small SDBs.

($\mathcal{A}_4$) Obtain an integration plan P by recursively partitioning the graph at the minimum cut. In our implementation, we used the push-relabel algorithm [12], which runs in $\mathcal{O}(|\mathcal{Q}|^3)$ time. As $|\mathcal{Q}|$ is generally a small number for typical database applications (e.g., the TPC-W implementation has only 15 parameterized queries), this step runs very fast. □

Phase $\mathcal{B}$. Executing parallel integration according to P at the required scale. This involves:

($\mathcal{B}_1$) SQP the input queries to get the SDBs at the required scale. This step is done *once* by the SQP engine in [7] for each input query.[8]

($\mathcal{B}_2$) Run SI operations in parallel according to P. As mentioned, this step involves (i) scanning the SDBs to construct the CBG, (ii) for each unpruned case, computing the satisfiable matching on the compressed flow network, and (iii) merging tuples according to the MSM and inserting them to form new SDBs. □

Phase $\mathcal{C}$. Data instantiating and quality checking. In this version, we use a top–down SDB instantiation approach. We

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

first instantiate the root SDB at the root of P and pose the queries on the instantiated database to check the quality (the relative error between the actual cardinality and the annotated cardinality of each operator). Note that the integration SDB at the root of P may not satisfy all queries' cardinality constraints. Therefore, if the instantiated DB at the root can satisfy all operators' cardinality constraints, then only one data instantiation (DI) operation is carried out; otherwise, additional DI operations are carried out on the intermediate SDBs—that also means more than one database would be generated. We remark that the top–down DI mechanism is yet another new improvement comparing with the previous version of MyBenchmark in [23]. Specifically, in [23], we used a bottom–up approach that instantiates every resulting SDB (and check the quality) right after each SI operation finishes. That results in unnecessary DI operations in case the integration is all the way good and only one database is needed for a workload. This improvement is significant because it can save the time of instantiating many intermediate SDBs, which is time consuming (e.g., about 12 min to instantiate a 10 GB SDB).

5 Experiments

We have carried out experiments using workloads extracted from four TPC benchmarks: TPC-C, TPC-E, TPC-H, and TPC-W. 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 20-node cluster with each node equipped with a Pentium Dual-Core 2.5 GHz PC with 8 GB memory. 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 the workload 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. After removing IUD and independent queries, the TPC-C workload, the TPC-E workload, and the TPC-W workload have 16, 15, and 15 queries, respectively. The TPC-H workload has 22 queries in total. But QAGen has not supported complicated SQL syntax such as "CASE/WHEN," "substring()" yet. So, we could only include 13 queries in the TPC-H workload (Q1, Q2, Q3, Q4, Q6, Q8, Q9, Q10, Q12, Q14, Q15, Q16, Q18, Q19). 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 data with scale factor 1, 5, and 10. In all experiments, we set the relative error tolerance to be 100 % for cardinalities in range [1, 1,000] (e.g., the acceptable range of cardinality 10 is [1, 20]; cardinality 0 is excluded), 10 % for cardinalities in range [1,001, 10,000] (e.g., the acceptable range of cardinality 5,000 is [4,500, 5,500]), and 1 % for cardinalities >10,001. During the phase of finding a good integration plan, the query cardinalities are down-scaled by the tool in [22] to a scale factor of 0.1. We set SI to accept satisfiable matchings whose size is better than $1 - T = 99\,\%$ of other unpruned cases with a probability of $Pr = 0.99$, resulting in at most 459 cases of being examined in each SI based on Equation 1. In all our experiments, a single database instance could be generated (integrated) for all four TPC workloads. So, we mainly evaluate the efficiency of MyBenchmark and its components.

5.1 Overall running time

Table 1 presents the overall and the breakdown of the running times of MyBenchmark for generating TPC workload-aware databases in different scales. We characterize the efficiency of MyBenchmark based on the items listed in Sect. 4. The rows with italics indicate those steps are carried out external tools (e.g., QAGen) and their running times are not our concern as long as they scale.

We observe that Phase $\mathcal{A}$, the plan search phase ($\sum_{\mathcal{A}}$), runs efficiently and an integration plan P could be found by within 1 min. The query cardinality scale-down tool [22] takes negligible time ($\mathcal{A}_1$). The SQP step carried out by the SQP engine in QAGen [7] takes some 10 s to generate small SDBs for every query in the workloads ($\mathcal{A}_2$). As TPC-H is the most complex workload, the SQP of its queries takes a longer time [7]. But that can still be finished within 1 min because we have 20 machines to run SQP in parallel. By running $C_2^{|\mathcal{Q}|}$ pairs of SI in parallel, the MSM summary graph can be obtained within 3 s for TPC-C, TPC-E, and TPC-W workloads and within 21 s for TPC-H workload. SIs in TPC-C, TPC-E, and TPC-W run faster than in TPC-H because they can often stop early (found a perfectly satisfiable matching) after examining some ten cases.

The integration plans found in Phase $\mathcal{A}$ are reused for generating data in all scale factors (SF=1, 10, 100) in Phase $\mathcal{B}$, the actual integration phase. We observe that this phase ($\sum_{\mathcal{B}}$) scales well among different scale factors. We also observe that the time of generating SDBs by SQP each input query ($\mathcal{B}_1$) dominates the time of the SI operations ($\mathcal{B}_2$) because of the effective optimization used in SI.

Phase $\mathcal{C}$, which instantiates the integrated SDB(s), is also as time expensive as step $\mathcal{B}_1$ (SQP time). That is because instantiating symbolic tuples requires calling an external constraint solver [7]. By having our new top–down instantiation methodology, however, only one (the root) SDB requires data instantiation.

Table 1 Execution times and breakdown (items $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{B}_1$ and $\mathcal{C}$ are carried out by external tools)

Item	Description/sub-item	TPC-C SF=0.1			TPC-E SF=0.1			TPC-H SF=0.1			TPC-W SF=0.1		
Phase $\mathcal{A}$: Finding a good integration plan P													
A_1	*Scale-down the input queries*	$< 1s$			$< 1s$			$< 1s$			$< 1s$		
A_2	*SQP all down-scaled queries once (in parallel)*	$9s$			$15s$			$22.8s$			$12s$		
$\mathcal{A}_3$	Build MSM summary by $C_2^{\|\mathcal{Q}\|}$ scaled-down SI (in parallel)	3 s			2.6 s			21 s			2.6 s		
$\mathcal{A}_4$	Deduce an integration plan P by recursive min-cut	<1 s			<1 s			<1 s			<1 s		
$\sum_{\mathcal{A}}$		13 s			18 s			45 s			14 s		
		SF=1.0	SF=10	SF=100	SF=1.0	SF=10	SF=100	SF=1.0	SF=10	SF=100	SF=1.0	SF=10	SF=100
Phase $\mathcal{B}$: Executing SI in parallel													
B_1	*SQP all queries in original scale (in parallel)*	$1.5min$	$14.7min$	$144min$	$2.5min$	$20.7min$	$201min$	$3.8min$	$38.2min$	$387min$	$2.0min$	$18.7min$	$181min$
$\mathcal{B}_2$	SI according to P (in parallel)	2 s	16 s	150 s	2 s	15.5 s	150 s	26 s	88 s	170 s	2.5 s	15 s	151 s
$\sum_{\mathcal{B}}$		1.6 min	15 min	147.2 min	2.6 min	21 min	204 min	3.9 min	40 min	391 min	2.1 min	18.9 min	185 min
Phase $\mathcal{C}$: Executing DI in parallel													
C	*Instantiate SDB top–down and cardinality checking*	$1.2min$	$12min$	$120min$	$1.3min$	$13.4min$	$130min$	$2.1min$	$22.7min$	$211min$	$1.1min$	$12min$	$121min$
Total: $\sum_{\mathcal{A}} + \sum_{\mathcal{B}} + \mathcal{C}$		2.8 min	27.3 min	268.2 min	4 min	34.6 min	335.2 min	6.1 min	63 min	613 min	3.2 min	32 min	306 min

Overall, MyBenchmark scales well with the data size and it takes about minutes to generate 1.0 GB workload-aware data and about 6 h to generate 100 GB data for complex TPC-H workloads in our 20-node cluster.

5.2 Varying workload size

To study the performance of MyBenchmark with respect to workloads of different sizes, we report the running times of MyBenchmark with different numbers of annotated queries as the input workload. In this experiment, we disable parallelization (i.e., the SI operations are executed one by one) such that we can observe the trend of running times with respect to the increasing number of queries without worrying about the parallel execution factor.

Figure 16 shows the results of generating scale factor 10 workload-aware database (whereas the plan finding phase is still at scale factor 0.1). Figure 16a shows that finding a good integration plan scales roughly linearly for TPC-C/E/W workloads and slightly poorer than linearly for TPC-H workload. The nonlinear factor comes from the building the MSM summary graph using $C_2^{|\mathcal{Q}|}$ SI operations (at scale factor SF 0.1). For TPC-C/E/W workloads, the nonlinear trend does not manifest very clearly because SQP is the dominating item (see $\mathcal{A}_2$ vs. $\mathcal{A}_3$ in Table 1). That trend however manifests in TPC-H workload because the SI operations in TPC-H workloads have running times close to SQP times (the reason is that TPC-H workload is highly complex; the SI operations in TPC-H workload need to examine more cases and seldom stop early; and details are presented in the next section). Nonetheless, that does not matter much because Phase $\mathcal{B}$ and Phase $\mathcal{C}$ actually dominate the overall execution time. Figure 16b shows that Phase $\mathcal{B}$ and Phase $\mathcal{C}$ together scale linearly with the workload size. So, Fig. 16c also shows the overall execution time scales linearly with the workload size.

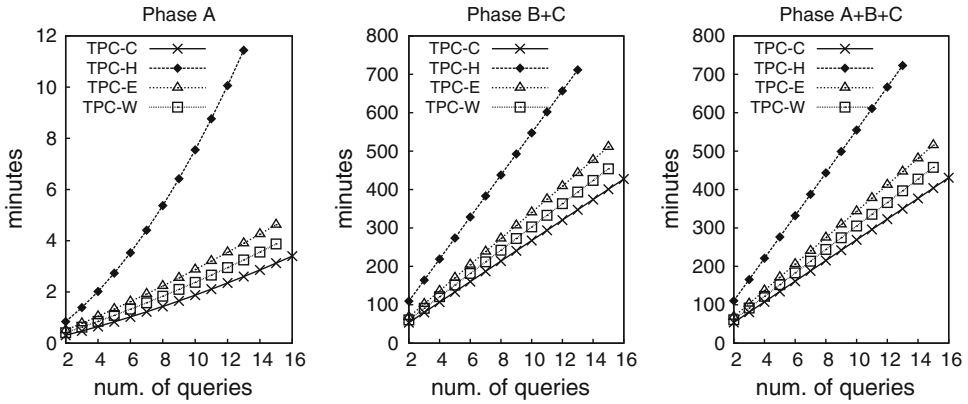

Fig. 16 Execution time versus workload size

Table 2 Effectiveness of *SI*

Details of *SI*	TPC-C	TPC-E	TPC-W	TPC-H
Maximum $\|R\|$	9	8	13	28
Maximum number of cases/CBGs to be examined by an SI in theory	512	256	8,192	268,435,456
Actual number of cases/CBGs examined when $\|R\|$ is maximum	22	8	24	459
Compression ratio (number of nodes in CBGs/number of nodes in compressed flow networks)	1,282:1	1,655:1	1,666:1	1,086:1

We remark that we can scale up the workload by only increasing the number of **real queries**. 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 directly scale up the number of unique relationships in a query workload because that depends on the query semantic. Nevertheless, as TPC benchmarks are simulating realistic workloads, we believe that scaling up real TPC queries is sufficient to observe the scalability of MyBenchmark with respect to workload size.

5.3 Efficiency and scalability of *SI*

We provide further information in Table 5.3 about the efficiency of the *SI* algorithm in this section.

We first discuss about the TPC-C, TPC-E, and TPC-W workloads. Independent of the scale factor, the maximum $|R|$ values found in TPC-C, TPC-E, and TPC-W are reported to be 9, 8, and 13, respectively. That can be interpreted as a total of 512 (TPC-C), 256 (TPC-E), and 8192 (TPC-W) cases have to be examined by *SI in theory*. However, it is reported that *SI* could find a perfectly satisfiable matching by examining only 22 cases in TPC-C, 8 cases in TPC-E, and 24 cases in TPC-W in our experiments when $|R|$ is maximum. That shows that our pruning techniques effectively prune the search space and the remaining cases are of high quality. Concerning the compression effectiveness, it is reported that the nodes in the compressed flow networks have only 31 nodes (TPC-C), 32 nodes (TPC-E), and 67 (TPC-W) nodes on average, which achieve a compression ratio of more than a thousand to one.

The maximum $|R|$ values of TPC-H is reported to be 28, meaning in theory a total of 268,435,456 cases have to be examined by *SI* in order to find the MSM. In our experiment, the *SI* operation under $|R| = 28$ case is reported to examine 459 cases, which means it could not early stop but examined all $s = 459$ cases under $Pr = 0.99$ and $T = 1\,\%$. Nonetheless, our experiments show that the MSMs found among those cases actually integrate all SDBs into one without violating the cardinality constraints.

In terms of running time, it is further reported that our implementation of *SI* can examine about 100 cases per second on average. An *SI* operation in TPC-H is reported to take about 5 s whereas an *SI* operation in TPC-C/E/W is reported to take about 0.5 s.

We do not carry out experiments to vary T and Pr because we know the integration quality could not be further improved by assigning values better than $T = 1\,\%$ and $Pr = 0.99$ (because all SDBs can be integrated into one already). It is also not insightful to assign values poor than $T = 1\,\%$ and $Pr = 0.99$ because the *SI* time, as shown in $\mathcal{B}_2$ in Table 1, after all, is dominated by the SQP time ($\mathcal{B}_1$) and the data instantiation time ($\mathcal{C}$). Therefore, there is almost no trade-off between the overall execution time and (poorer) values for T and Pr. However, we remark that the optimizations and approximations in *SI* are still very important. Without them, an *SI* in TPC-H has to examine 268,435,456 CBGs in the worst case and that would need about 31 days per *SI* (based on the empirical results of processing 100 CBGs requires 1 s).

Finally, we can see from Table 1 that *SI* scales well across different scale factors ($\mathcal{B}_2$). Take TPC-H as an example. It is reported that the integration plan of TPC-H consists of four levels, so the 26 s listed in Table 1 (SF=1.0) is attributed to 4 *SI*s operations, with each takes about 6.5 s. Among that 6.5 s, it is reported that about 5 s are spent on matching several hundreds of CBGs and about 1.5 s are I/Os that spent on scanning the SDBs and inserting integrated tuples to new SDBs. Now, the *SI* time at SF 10 that takes 88 s can be explained as follows. First, the time of scanning SDBs and writing the new SDBs are scaled up by a factor of 10, resulting in about 15 s of I/Os. As $|R|$ is independent of the scale, the time of examining the CBGs in *SI* is still about 5 s for TPC-H workload. So, the time of an *SI* at this scale factor is about $15 + 5 = 20$ s. As there is a total of 4 levels of *SI* in TPC-H's integration plan, the whole integration should take about 4×20 s, which roughly matches to the reported numbers, 88 s.

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. Our experiments show that MyBenchmark can efficiently generate high-quality workload-aware databases for TPC-C, TPC-E, TPC-H, and TPC-W workloads using a cluster of machines in parallel.

7 Appendix: Proofs

7.1 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 [14] takes as input a set of elements $\mathcal{S} = \{\mathcal{S}_\infty, \mathcal{S}_\in, \ldots, \mathcal{S}_{\ni\setminus}\}$ and a collection of 3-element set $\mathcal{C} = \{\mathcal{C}_\infty, \mathcal{C}_\in, \ldots, \mathcal{C}_{\Updownarrow}\}$ 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.

7.2 Proof sketch of the optimal integration plan problem

Proof Sketch. Given an instance of a cross product optimization [13], 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. □

References

1. Dbunit. http://www.dbunit.org
2. DTM Data Generator. http://www.sqledit.com/dg/
3. IBM DB2 Test Database Generator. http://www-306.ibm.com/software/data/db2imstools/db2tools/db2tdbg/
4. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice-Hall Inc, Upper Saddle River, NJ (1993)
5. Arasu, A., Kaushik, R., Li, J.: Data generation using declarative constraints. In: SIGMOD Conference, pp. 685–696 (2011)
6. Binnig, C., Kossmann, D., Lo, E.: Reverse query processing. In: ICDE (2007)
7. Binnig, C., Kossmann, D., Lo, E., Ozsu, M.T.: QAGen: generating query aware test databases. In: SIGMOD (2007)
8. Bruno, N., Chaudhuri, S.: Flexible database generators. In: VLDB, pp. 1097–1107 (2005)
9. Bruno, N., Chaudhuri, S., Thomas, D.: Generating queries with cardinality constraints for DBMS testing. In: TKDE (2006)
10. Chays, D., Deng, Y., Frankl, P.G., Dan, S., Vokolos, F.I., Weyuker, E.J.: An AGENDA for testing relational database applications. Software testing, verification and reliability (2004)
11. Chen, M.-S., Yu, P.S., Wu, K.-L.: Optimization of parallel execution for multi-join queries. IEEE Trans. Knowl. Data Eng. **8**, 416–428 (1996)
12. Cherkassky, B.V., Goldberg, A.V.: On implementing the push-relabel method for the maximum flow problem. Algorithmica **19**(4), 390–410 (1997)
13. Cluet, S., Moerkotte, G.: On the complexity of generating optimal left-deep processing trees with cross products. In: ICDT, pp. 54–67 (1995)
14. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co. New York, NY, USA (1990)
15. Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., Jacobsen, H.-A.: Bigbench: towards an industry standard benchmark for big data analytics. In: SIGMOD Conference, pp. 1197–1208 (2013)
16. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly generating billion-record synthetic databases. In: SIGMOD, pp. 243–252 (1994)
17. Haftmann, F., Kossmann, D., Lo, E.: A framework for efficient regression tests on database applications. VLDB J. **16**(1), 145–164 (2007)
18. He, Z., Kunjir, M., Lim, H., Lo, E., Babu, S., Hsu, M., Castellanos, M.: Benchmarking "no one size fits all" big data analytics. In: The Third Workshop on Big Data Benchmarking (2013)
19. Hoag, J.E., Thompson, C.W.: A parallel general-purpose synthetic data generator. SIGMOD Rec. **36**(1), 19–24 (2007)
20. Houkjær, K., Torp, K., Wind, R.: Simple and realistic data generation. In: VLDB, pp. 1243–1246 (2006)
21. Kini, A., Shankar, S., Naughton, J.F., DeWitt D.J.: Database support for matching: limitations and opportunities. In: SIGMOD Conference, pp. 85–96 (2006)
22. Lo, E., Binnig, C., Kossmann, D., Ozsu, M.T., Hon, W.-K.: A framework for testing DBMS features. VLDB J. **19**(2), 203–230 (2010)
23. Lo, E., Cheng, N., Hon, W.-K.: Generating databases for query workloads. PVLDB **3**(1), 848–859 (2010)
24. Mannila, H., Räihä, K.-J.: Test data for relational queries. In: PODS, pp. 217–223 (1986)
25. Mishra, C., Koudas, N., Zuzarte, C.: Generating targeted queries for database testing. In: SIGMOD, pp. 499–510 (2008)
26. Neufeld, A., Moerkotte, G., Lockemann, P.C.: Generating consistent test data for a variable set of general consistency constraints. VLDB J. **2**(2), 173–213 (1993)
27. Olston, C., Chopra, S., Srivastava, U.: Generating example data for dataflow programs. In: SIGMOD Conference, pp. 245–256 (2009)

28. Rabl, T., Poess, M.: Parallel data generation for performance analysis of large, complex rdbms. In: DBTest 2011, p. 5 (2011)
29. Sellis, T.K.: Multiple-query optimization. ACM Trans. Database Syst. **13**(1), 23–52 (1988)
30. Stephens, J.M., Poess, M.: Mudd: a multi-dimensional data generator. In: WOSP, pp. 104–109 (2004)
31. Tay, Y.: Data generation for application-specific benchmarking. In: PVLDB, p. 4 (2011)

View publication stats