The VLDB Journal
DOI 10.1007/s00778-009-0157-y

REGULAR PAPER

A framework for testing DBMS features

Eric Lo · Carsten Binnig · Donald Kossmann · M. Tamer Özsu · Wing-Kai Hon

Received: 12 July 2008 / Revised: 11 March 2009 / Accepted: 7 July 2009

Abstract Testing a specific feature of a DBMS requires controlling the inputs and outputs of the operators in the query execution plan. However, that is practically difficult to achieve because the inputs/outputs of a query depend on the content of the test database. In this paper, we propose a framework to test DBMS features. The framework includes a database generator called QAGen so that the generated test databases are able to meet the test requirements defined on the test queries. The framework also includes a set of tools to automate test case constructions and test executions. A wide range of DBMS feature testing tasks can be facilitated by the proposed framework.

Keywords Database testing · Data generation · Symbolic query processing · Symbolic execution

This work is supported by GRF grant PolyU 5250/09 from Hong Kong RGC and ICRG grants (Project Numbers: 1-ZV5R and A-PC0N) from The Hong Kong Polytechnic University.

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

C. Binnig · D. Kossmann
ETH Zurich, Zurich, Switzerland

M. Tamer Özsu
David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada

W.-K. Hon
Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan

1 Introduction

The complexity of database management systems (DBMS) makes the addition of new features or the modifications of existing features difficult. The impact of the modifications on system performance and on other components is hard to predict. Therefore, after each modification, it is necessary to run tests to validate the system correctness and evaluate the relative system improvements under a wide range of scenarios and workloads.

Today, a common methodology for testing a database system is to generate a comprehensive set of test databases and then study the before-and-after system behavior by executing many test queries over those test databases. Test databases can be generated by database generation tools such as IBM DB2 Database Generator [23], DTM Data Generator [13], MUDD [35], as well as some research prototypes (e.g., [4,19,21]). These tools allow a user to define the sizes and the data characteristics (e.g., value distributions and inter/intra-table correlations) of the base tables. The next step is to manually create test queries, or stochastically generate many valid test queries using query generation tools such as RAGS [34] or QGEN [32]. The test queries are then posed on the generated test databases in order to test the various system components such as testing the query optimizer [14] and the query parser [7].

Unfortunately, the current testing methodology is inadequate for testing individual features of database systems. To do so, it is necessary to control the input/output of query operators during a test. Consider the following example. Assume a testing team for a DBMS product wants to test how a newly designed memory manager influences the correctness and/or the performance of multi-way hash join queries (i.e., how the per-operator memory management strategy of the memory manager affects the resulting execution plans). In order to

Published online: 01 August 2009

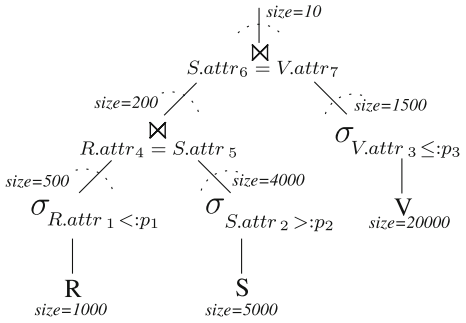

Fig. 1 A test case: a query with operator constraints

do so, a test case like the one shown in Fig. 1 is designed (the figure is modified from [5]). A test case T is a parameterized query Q with a set of constraints C defined on the query operators. In Fig. 1, the test query of the test case first joins a large filtered table S with a filtered table R to get a small join result. Then, the small intermediate join result is joined with a filtered table V to obtain a small final result. Since the memory requirements of a hash join depends on the size of its inputs, it would be beneficial if the input/output of each individual operator in the query tree could be controlled/tuned according to the test requirements [5]. For example, the memory allocated to $\bowtie_{S.attr_6=V.attr_7}$ by the memory manager can be studied by controlling the output cardinality of $\sigma(R) \bowtie \sigma(S)$ and the output cardinality of $\sigma(V)$. Unfortunately, even though the tester can instruct the database engine to evaluate the test query with a specific physical execution plan (e.g., fixing the join order and forcing the use of hash-join as the join algorithm) [5], it is not easy to control the properties of (intermediate) results (e.g., the output cardinality of $\bowtie_{S.attr_6=V.attr_7}$) as those properties rely on the *content* of the test database.

In DBMS feature testing, test queries are usually designed by testers and executed on some test databases. The test databases used in testing should *cover* the test cases, i.e., if the test query Q (with parameter values P) specified in T is executed on D (denoted by $Q_P(D)$), the (intermediate) query results of Q should meet the constraints C defined in T. Unfortunately, existing test database generators do not take test cases as input. As a result, the test databases generated by those tools rarely cover the test cases. For example, it is hard to find a test database to cover the test case in Fig. 1, which expects the output cardinality to be 10 tuples, unless the database content is *manually tuned*.

Recently, Bruno et al. [5] and Mishra et al. [30] view testing DBMS features as a *targeted query generation* (TQG) problem. Given a test database D, a parameterized conjunctive query Q, and cardinality constraints C over the sub-expressions of Q, [5,30] discuss how to find the set of parameter values P of Q such that the output cardinality of each operator in Q meets C. The TQG problem is $\mathcal{NP}$-hard [30]. Therefore, [5,30] develop approximation solutions that find P for Q such that the output cardinality of each operator approximately (instead of exactly) meets C. Although useful, their approaches are limited by the given test databases. In particular, given a predefined test database (e.g., say, an empty test database in extreme case), there may be no parameter values that permit the query results of Q to approximately meet the constraints defined in the test case. Even if the given test database does cover the test case, as the solution space is too large, solutions in [5,30] are restricted to supporting simple SPJ queries with single-sided predicates (e.g., $p_1 \leq a$ or $a \leq p_2$) or double-sided predicates (e.g., $p_1 \leq a \leq p_2$) (where a is an attribute and p_1 and p_2 are parameter values). That limits the construction of more realistic test queries which include operators like grouping and aggregation and more complicated expressions.

We observe that the test database generation process is the main culprit of ineffective DBMS feature testing. Currently, test databases are generated without taking the test cases as input. Thus, executing a test query on top of such generated databases does not guarantee that the expected (intermediate) query results can be obtained. Therefore, the only way to carry out testing meaningfully is to do a painful trial-and-error test database generation process. That is, in order to execute a test case T, we generate many test databases, or manually tune the content of the generated test databases, until a suitable test database is found.

In this paper, we address the DBMS feature testing problem in a different and novel way. We propose a DBMS feature testing framework which consists of a set of tools. One of them is a test database generator called QAGen. QAGen is a "Query-Aware" test database generator which generates query-aware test databases for each individual test case. It takes as input a database schema M and a test case T (with a set of constraints C defined on the base tables and optionally defined on the operators) and generates a query-aware database instance D and query parameter values P such that D satisfies M and the (intermediate) results of $Q_P(D)$ (closely) meets C.

As QAGen considers the test cases as first-class citizens during the data generation process, the generated database instances are able to cover a broader class of test queries (e.g., complex TPC-H queries) and useful for a variety of DBMS feature testing tasks. For example, testers can use QAGen to generate a test database that controls the size of the intermediate join results. With such a test database, they can test the accuracy of the cardinality estimation components (e.g., histograms) inside a query optimizer by fixing the join order.[1]

[1] However, QAGen is not designed to test the join reordering feature of a query optimizer directly because in this case the *physical* join ordering should not be fixed by the tester; and the intermediate cardinalities guaranteed by QAGen may affect the optimizer. This can result in a different physical execution plan with different intermediate results.

As another example, testers can use QAGen to generate a test database that guarantees the input and the output sizes (the number of groups) for a GROUP-BY operator. With such a test database, we can evaluate the performance of the grouping operation under a variety of cases such as in multi-way join queries or in nested queries.

In addition to QAGen, the framework also includes a set of tools to automate the steps of DBMS test case construction and test case execution. The framework automatically creates and executes a set of test cases that satisfy a certain test coverage. As a result, testers do not need to worry how to determine the constraints of a test query (e.g., the cardinality constraint "$size = 500$" in Fig. 1) in order to form useful test cases.

The contributions of this paper are summarized as follows:

- A DBMS testing framework is presented. The framework facilitates the tasks of testing various DBMS features. It includes a query-aware test database generator QAGen, which is an extension of the one in our earlier work [3]. This version of QAGen has been extended to support all SQL operators (under a few restrictions) such that a larger class of SQL queries can be supported.
- The implementations of QAGen are presented. New algorithms are introduced in order to improve the performance of some inefficient operations in [3].
- A DBMS test case generation algorithm is presented. The algorithm automates the tasks of manually constructing meaningful DBMS test cases. It is implemented as part of the framework. Consequently, the framework can automatically evaluate the features of a DBMS and generate meaningful test reports.
- The efficiency and the effectiveness of the proposed framework are studied through comprehensive experiments and the detailed experimental results are presented.

The remainder of this paper is organized as follows: Sect. 2 gives an overview of QAGen. Section 3–5 describe the architecture of QAGen and the algorithms used. Section 6 presents the testing framework that is built on top of QAGen. Section 7 presents the experimental results. Section 8 discusses related work. Section 9 contains conclusion and suggestions for future work.

2 QAGen

In this section, we introduce QAGen, which is the most important component of our DBMS testing framework. QAGen is a query-aware test database generator that takes as input a query, a specification of constraints on intermediate query results (e.g., cardinalities and value distributions), and a specification of the database schema and generates as output a query-aware test database. We define the problem **Query-Aware Test Database Generation** as follows:
Given a database schema M (where SQL data types are bounded), a test case T consists of a parameterized SQL query Q (expressed in the form of a relational algebra expression) and a set of user-defined constraints C defined on operator(s) of Q. Find a database instance D and the set of parameter value(s) P such that D satisfies M and $Q_P(D)$ meets constraints C.

A decision problem can be constructed based on the problem statement above which asks whether or not a database instance D exists given T and M. If the set C does not contain contradicting constraints, we can always find a D by exhaustively trying all possible database instances because the data types in SQL are bounded.[2] In practice, however, the hardness of the problem depends on the complexity of the given Q, C and M. For example, in most cases QAGen can generate a test database D such that $Q_P(D)$ meets C *exactly*. However, in some cases, QAGen also considers approximate solutions because the generation process involves solving a weakly $\mathcal{NP}$-complete problem. Nonetheless, approximate solutions is widely acceptable in DBMS testing [5,30]. For instance, to test the memory manager of a DBMS, it does not matter whether the final join result in Fig. 1 contains exactly 10 tuples or 11 tuples.

2.1 Overview

QAGen is designed for experienced testers as a test automation tool. Multiple copies of QAGen can be run on the machines of a test farm in order to generate multiple test databases simultaneously. The test cases together with the generated databases can form a large scale regression test suite of a DBMS product.

In contrast to traditional database generators which only allow testers to specify constraints on the base tables (a tester thus cannot specify operator constraints, say, the output cardinality of a join in an intrinsic way), QAGen allows a tester to annotate constraints on both operators and base tables, and thus the testers can easily get a meaningful test database for a distinct test case. QAGen is also designed to be extensible so that it can easily incorporate new operator constraints. For example, sometimes it would be advantageous to add new kinds of constraints to an operator in addition to the cardinality constraint during testing. For instance, the GROUP-BY operator may need to control not only the output size (i.e., the number of groups), but also how to distribute the input

[2] A more general problem statement may be constructed if we do not restrict to SQL (data types). In that case, the problem is undecidable because it is undecidable to find solutions for arbitrary predicates (a reduction from Hilbert's tenth problem).

	a	b
$t1$:	$\$a1$	$\$b1$
$t2$:	$\$a2$	$\$b2$

Table R

	c	d
$t3$:	$\$c1$	$\$d1$
$t4$:	$\$c2$	$\$d2$
$t5$:	$\$c3$	$\$d3$
$t6$:	$\$c4$	$\$d4$

Table S

	a	$b=c$	d
$t7$:	$\$a1$	$\$b1$	$\$d1$
$t8$:	$\$a1$	$\$b1$	$\$d2$
$t9$:	$\$a1$	$\$b1$	$\$d3$
$t10$:	$\$a2$	$\$b2$	$\$d4$

$R \bowtie_{b=c} S$

	e	f
$t11$:	$\$e1$	$\$f1$
$t12$:	$\$e2$	$\$f2$

Table V

Fig. 2 Example of symbolic query processing $(R \bowtie_{b=c} S) \bowtie_{a=e} V$

to the predefined output groups (i.e., some groups have more tuples while others have fewer).

QAGen generates query-aware test databases in two phases: (1) the symbolic query processing (SQP) phase, and (2) the data instantiation phase. The goal of the SQP phase is to capture the user-defined constraints on the query into the target database. To process a query without concrete data, QAGen integrates the concept of symbolic execution [25] from software engineering into traditional query processing. Symbolic execution is a well known program verification technique, which represents values of program variables with symbolic values instead of concrete data, and manipulates expressions based on those symbolic values. Borrowing this concept, QAGen first instantiates a database which contains a set of symbols instead of concrete data (thus the generated database in this phase is called a *symbolic database*). Fig. 2 shows an example of a symbolic database with three *symbolic relations* R, S and V. A symbolic relation is a table that consists of a set of *symbolic tuples*. Inside each symbolic tuple, the values are represented by *symbols* rather than by concrete values, e.g., symbol $\$a1$ in symbolic relation R in Fig. 2 represents any value under the domain of attribute a. The formal definition of these terms is given in Sect. 4. For the moment, let us just treat the symbolic relations as normal relations and treat the symbols as variables. Since the symbolic database is a generalization of relational databases and provides an abstract representation for concrete data, this allows QAGen to control the output of each operator of the query.

The SQP phase leverages the concept of traditional query processing. First, the input query is analyzed by a *query analyzer*. Then, users specify their desired constraints on the operators of the query tree. Afterwards, the input query is executed by a *symbolic query engine* as in traditional query processing; i.e., each operator is implemented as an iterator, and the data flows from the base tables up to the root of the query tree [18]. However, unlike traditional query processing, the symbolic execution of operators deals with symbolic data rather than concrete data. Each operator manipulates the input symbolic data according to the operator's semantics and user-defined constraints, and incrementally imposes the constraints defined on the operators to the symbolic database. After this phase, the symbolic database is a query-aware database that meets all the constraints defined in the test case (but without concrete data).

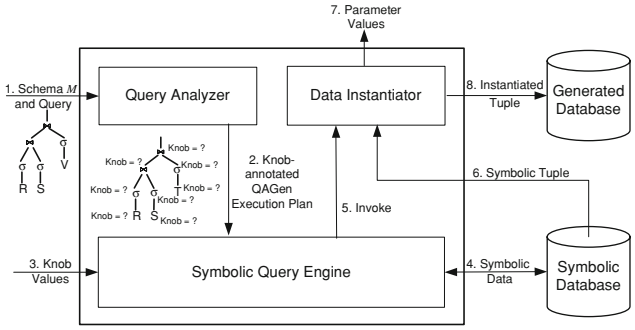

Fig. 3 QAGen architecture

The data instantiation phase follows the SQP phase. This phase reads the tuples from the symbolic database that are prepared by the SQP phase and instantiates the symbols with real data values. The instantiated tuples are then inserted into the target database.

To allow a user to define different test cases for the same query, the input query of QAGen is in the form of a relational algebra expression. For example, if the input query is a three-way join query $(\sigma_{age>p_1} Customer \bowtie Orders) \bowtie Lineitem$, then users can specify a join key distribution (e.g., a Zipf distribution) between the lineitems and the orders that join with customers with an age greater than p_1. On the other hand, if the input query is $(Orders \bowtie Lineitem) \bowtie \sigma_{age>p_1} Customer$, then users can specify the join key distribution between all orders and all lineitems.

Figure 3 shows the general architecture of QAGen. It consists of the following components: a Query Analyzer, a Symbolic Query Engine, a Symbolic Database and a Data Instantiator.

2.2 Query analyzer

In the beginning of the SQP phase, QAGen first takes as input a parameterized query Q and the database schema M. The query Q is then analyzed by the query analyzer component in QAGen. The query analyzer has two functionalities:

(1) Knob annotation. The query analyzer analyzes the input query and determines which *knob(s)* are available for each operator. A knob can be regarded as a parameter of an operator that controls the output. A basic knob that is offered by QAGen is the output cardinality.[3] This knob allows a user to control the output size of an operator. However, whether such a knob is applicable depends on the operator and its input characteristics.

Figure 4 shows the knobs of each operator offered by QAGen under different cases. As an example, for a simple aggregation query `SELECT MAX(a) FROM R`, the cardi-

[3] The output cardinality of an operator can be specified as an absolute value or as a selectivity. Both ways are equivalent.

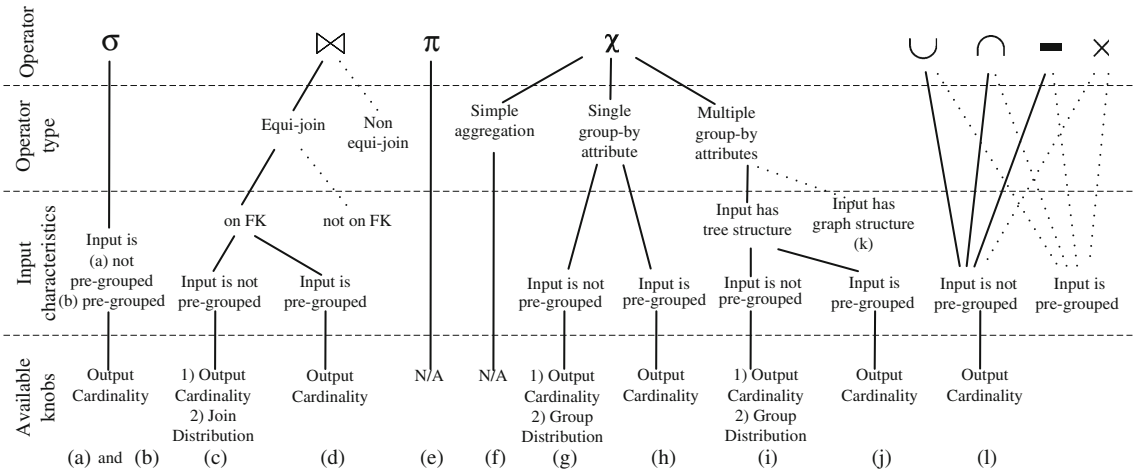

Fig. 4 Symbolic query processing framework of QAGen

nality constraint knob should not be available for the aggregation operator (χ). This is because the output cardinality of `MAX(a)` is always one (Fig. 4 case (f)). As another example, the available knob(s) of an equi-join ($\bowtie$) depend on whether the input is *pre-grouped* or not on the join keys. If the input is pre-grouped, the equi-join can only offer the output cardinality as knob (Fig. 4 case (d)). If the input is not pre-grouped, users are allowed to tune the join key distribution as well (Fig. 4 case (c)). The input of an operator is *pre-grouped* w.r.t. an attribute a if and only if there is at least one symbol which is not distinct in a (Sect. 3 gives formal definitions of all input characteristics). Consider a three-way join query $(R \bowtie_{b=c} S) \bowtie_{a=e} V$ on the three symbolic relations R, S, and V in Fig. 2. When symbolic relation R first joins with symbolic relation S on attributes b and c, it is possible to specify the join key distribution such as joining the first tuple $t1$ of R with the first three tuples of S (i.e., $t3$, $t4$, $t5$); and the last tuple $t2$ of R joins with the last tuple $t6$ of S (kind of like Zipf distribution [37]). However, after the first join, the intermediate join result $R \bowtie_{b=c} S$ is pre-grouped w.r.t. attributes a, b and c (e.g., symbol $\$a1$ is not distinct on attribute a in the join result). Therefore, if this intermediate join result further joins with symbolic relation V on attributes a and e, then the distribution cannot be freely specified by a user. This is because if the first tuple $t11$ of V joins with the first tuple $t7$ of the intermediate results, this implies that $\$e1 = \$a1$ and thus $t11$ must join with $t8$ and $t9$ as well.

This example shows that it is necessary to analyze the query in order to annotate the proper knobs to the operators. For this purpose, the query analyzer analyzes the input query in a bottom-up manner (i.e., starting from input schema M) and incrementally pre-computes the output characteristics of each operator (e.g., annotates an attribute of the output of an operator as pre-grouped if necessary). In the example, the query analyzer realizes that the intermediate join result $R \bowtie_{b=c} S$ is pre-grouped w.r.t. attributes a, b and c. Based on this information, the query analyzer disables the join key distribution knob on the next equi-join that joins with V. Thus, the query analyzer annotates the appropriate knob(s) to each operator according to Fig. 4. The output of the query analyzer is an annotated query tree with the appropriate knob(s) on each operator. Section 3 presents the details of this step.

Figure 4 is also a summary of the class of SQL queries that QAGen supports. Comparing with the previous version of QAGen in [3], this version of QAGen supports all SQL operators except the Cartesian product operation($\times$), which in practice is rarely used. The dotted lines show some special cases that the current version of QAGen does not support. According to Fig. 4, the current version of QAGen already suffices to cover 14 out of 22 complex TPC-H queries. In this paper, we assume that the number of possible values in the domain of a GROUP-BY attribute is greater than the number of tuples to be output for an aggregation operator.

(2) Assign physical implementations to operators. As shown in Fig. 4, different knobs are available under different input characteristics. In general, different (combinations of) knobs of the same operator need separate implementation algorithms. Moreover, even for the same (combination of) knobs of the same operator, different implementation algorithms are conceivable (this is akin to traditional query processing where an equi-join operation can be done by hash-join or sort-merge join). Consequently, the second purpose of the query analyzer is to assign an implementation to each operator and return a knob-annotated query execution plan. Section 4 presents the implementation algorithms for each operator in QAGen.

In general, the job of the query analyzer is analogous to the job of the query optimizer in traditional query processing. However, in the current version of QAGen, only one implementation algorithm for each (combination of) knob is

available. If there is more than one possible implementation for a knob (or a combination of knobs), the query analyzer can be extended to be a query optimizer, thereby selecting the most appropriate implementation based on the estimated query processing time or other quality criteria.

Supporting new variants of an operator (e.g., theta join) or adding new knobs (which may depend on new input characteristics) to an operator is straightforward in QAGen. For example, adding a new knob to an operator can be done by incorporating the corresponding implementation of the operator into the symbolic query engine and then updating the query analyzer about the input characteristics that this new knob depends on.

2.3 Symbolic query engine and database

The symbolic query engine of QAGen is the heart of the SQP phase and it is similar to a traditional database query engine. It interprets a knob-annotated query execution plan given by the query analyzer. The symbolic query engine also uses an iterator model. That is, every operator is implemented as an iterator. Each iterator consumes symbolic tuples from it child iterator(s) one-by-one and returns symbolic tuples to its parent iterator.

Before the symbolic query engine starts execution, the user can specify the value(s) for the available knob(s) of each operator in the knob-annotated execution plan. It is acceptable to specify values for only a few knobs. If the value of a knob is not specified, it would be determined according to the rules given by the creator of the knob (however, base table sizes must be given by the tester).

Similar to traditional query processing, most operators in SQP can be processed in a pipelined mode, but some cannot. For example, the equi-join operator in SQP is a blocking operator under certain circumstances. In these cases, the symbolic query engine materializes the intermediate results into the symbolic database if necessary. In SQP, a table in a query tree is regarded as an operator. During its open() method, the table operator initializes a symbolic relation based on the input schema M and the user-defined constraints (e.g., table sizes) on the base tables.

In SQP, an operator evaluates the input tuples according to its own semantics. On the one hand, it imposes additional constraints to each input tuple in order to reflect the constraints defined on the operator. On the other hand, it controls its output to its parent operator so that the parent operator can work on the right tuples. As a simple example, assume the input query is a simple selection query $\sigma_{a \geq p_1} R$ on symbolic relation R in Fig. 2 and the user specifies the output cardinality as 1 tuple. If the getNext() method of the selection operator iterator is invoked, it reads tuple $t1$ from R, annotates a *positive* constraint $[\$a1 \geq p_1]$ (i.e., the selection predicate) to symbol $\$a1$ and returns tuple $\langle \$a1, \$b1 \rangle$ to its parent. When the getNext() method of the selection operator is invoked a second time, it reads the next tuple $t2$ from R, and annotates a *negative* constraint $[\$a2 < p_1]$ (i.e., the negation of the selection predicate) to symbol $\$a2$. However, this time, it does not return this tuple to its parent. That is because the cardinality constraint (1 tuple) is already met.

Note that, although we assume the users of QAGen are experienced testers, it still possible that they accidentally specify some contradicting knob values on test cases. For instance, a tester may accidentally specify the output cardinality of the selection in the above example as 10 tuples even if she specified table R to have only two tuples. In these cases, QAGen will return corresponding error messages for the tester to correct the test case.

2.4 Data instantiator

The data instantiation phase starts after the SQP phase. The data instantiator reads the symbolic tuples from the symbolic database and instantiates the symbols inside each symbolic tuple by a constraint solver. In QAGen, we treat the constraint solver as an external black box component which takes as input a constraint formula (in propositional logic) and returns a possible instantiation on each variable. For example, if the input constraint formula is $40 < \$a1 + \$b1 < 100$, then the constraint solver may return $\$a1 = 55$, $\$b1 = 11$ as output (or any other correct instantiation). Once the data instantiator has collected all the concrete values for a symbolic tuple, it inserts a corresponding tuple (with concrete values) into the target table.

3 Query analyzer in QAGen

This section presents the details of the query analyzer in QAGen. Given the input relational algebra expression, the query analyzer serves two purposes: (1) annotates proper knobs to the query operators, and (2) assigns physical implementations to operators so as to form a physical execution plan. QAGen currently supports only one physical implementation for each possible combination of knobs per relational algebra operator. As a result, (2) is straightforward and for brevity we omit details of this step. This section focuses on (1), i.e., how to analyze the query and determine the available knob(s) for each operator in the input query.

The query analyzer determines the input characteristics of each operator of the input relational algebra expression in order to decide what kinds of knobs are available for each operator. In SQP, there are four types of input characteristics: *pre-grouped*, *not pre-grouped*, *tree-structure*, and *graph-structure*. Let A be the set of attributes of the input of an operator. The input characteristics are defined as follows:

Definition Pre-grouped/Not pre-grouped: The input of an operator is *not pre-grouped* with respect to an attribute $a \in A$, iff there is a functional dependency $a \rightarrow A$ (which means that a is distinct) that holds in the input. Otherwise, the input of the operator is *pre-grouped* with respect to attribute a.

Definition Tree-structure/Graph-structure: A set of attributes $A' \subset A$ of the input of an operator has a *tree-structure*, iff either the functional dependency $a_i \rightarrow a_j$ or $a_j \rightarrow a_i$ holds in the input of the operator for all a_i, a_j in A' and $a_i \neq a_j$. Otherwise, the set of attributes $A' \subset A$ of the input of the operator has a *graph-structure*.

Since the definition of the input characteristics of an operator solely depends on the functional dependencies, the type of knobs available for an operator can be easily determined according to Fig. 4. In particular, the query analyzer can compute the set of functional dependencies that holds in each intermediate result in a bottom-up fashion using the rules in [2] (i.e., starting from the base tables). Note that, as it is not possible to derive the complete set of functional dependences that hold on the (intermediate) results of a relational algebra expression containing the MINUS operator [26], QAGen might offer wrong knobs to some operators above the MINUS operators in some cases. In this cases, QAGen will warn the users to check the knobs before it starts processing. Nonetheless, this problematic case rarely happens in practice. In our experiments of processing 14 TPC-H queries, QAGen offers correct knobs to all operators.

We reuse the query $(R \bowtie_{b=c} S) \bowtie_{a=e} V$ in Fig. 2 to illustrate the pre-grouped and not pre-grouped data characteristics. In Fig. 2, the intermediate result $R \bowtie_{b=c} S$ is pre-grouped w.r.t. attributes a, b and c (where $b = c$) and is not pre-grouped w.r.t. attribute d. This is because:

- As we will see in the next section, all symbols in the base tables are initially distinct (see tables R and S in Fig. 2 as an example). Therefore, the set of non-trivial functional dependencies of R is $\{a \rightarrow b, b \rightarrow a\}$ and the set of functional dependencies of S is $\{c \rightarrow d, d \rightarrow c\}$.
- According to the functional dependency calculation rule for joining in [2], the join predicate $b = c$ (with c as a foreign-key of b) creates a functional dependency $c \rightarrow b$. Therefore, the set of functional dependencies of the intermediate join result $R \bowtie_{b=c} S$ is $\{a \rightarrow b, b \rightarrow a, c \rightarrow d, d \rightarrow c, c \rightarrow b, c \rightarrow a, d \rightarrow b, d \rightarrow a\}$.
- Among the set of attributes $A : \{a, b, c, d\}$ of $R \bowtie_{b=c} S$, attributes c and d functionally determines attributes a and b. As a result, according to the definition of pre-grouping, the intermediate result is not pre-grouped w.r.t. c and d but is pre-grouped w.r.t. a and b.

We use another example to illustrate the concept of tree-structure and graph-structure input characteristics. Assume the following table is an intermediate result of a query:

a	b	c	d
$\$a1$	$\$b1$	$\$c1$	$\$d1$
$\$a2$	$\$b1$	$\$c1$	$\$d2$
$\$a3$	$\$b2$	$\$c1$	$\$d2$
$\$a4$	$\$b3$	$\$c2$	$\$d1$

Assume the following functional dependencies hold in this intermediate result: $a \rightarrow \{b, c, d\}$, and $b \rightarrow \{c\}$. Following the definitions of tree and graph structure, the attribute set $A = \{a, b, c\}$ has a tree-structure because all attributes are functional dependent on each other. On the other hand, the attribute set $A = \{a, b, d\}$ has a graph-structure because there is no functional dependency between b and d (i.e., neither $\{b\} \rightarrow \{d\}$, nor $\{d\} \rightarrow \{b\}$ holds in the intermediate result).

4 Symbolic query engine in QAGen

This section presents the details of the symbolic query engine in QAGen. First, we define the data model of symbolic data and discuss how to physically store the symbolic data. Then, we present algorithms for implementing the symbolic executions of operators through a running example.

4.1 Symbolic data model

4.1.1 Definitions

A *symbolic relation* consists of a *relational schema* and a *symbolic relation instance*. The definition of a relational schema is the same as the classical definition of a relational schema [9]. Let $R(a_1{:}dom(a_1), \ldots, a_i{:}\ dom(a_i), \ldots, a_n{:}\ dom(a_n))$ be a relational schema with n attributes; and for each attribute a_i, let $dom(a_i)$ be the domain of attribute a_i.

A symbolic relation instance is a collection of *symbolic tuples* T (using bag semantics; in order to support SQL). Each symbolic tuple $t \in T$ is a n-tuple with n *symbols*: $\langle s_1, s_2, \ldots, s_n \rangle$. As a shorthand, symbol s_i in tuple t can be referred by $t.a_i$. A symbol s_i is associated with a set of *predicates* P_{s_i} (N.B. P_{s_i} can be empty). The value of symbol s_i represents any one of the values in the domain of attribute a_i that satisfies *all* predicates in P_{s_i}. The symbols s_i and s_j represent different values if $i \neq j$ and they are from the same attribute. A predicate $p \in P_{s_i}$ of a symbol s_i is a propositional formula that involves at least s_i and zero or more other symbols that appear in different symbolic relation instances. Therefore, a symbol s_i with its predicates P_{s_i} can be represented by a conjunction of propositional logic formulas. A *symbolic database* is defined as a set of symbolic relations

and there is a one-to-many mapping between one symbolic database and many traditional relational databases.

4.1.2 Data storage

Symbolic databases are a generalization of relational databases and provide an abstract representation of concrete data. Given the close relationship between relational databases and symbolic databases, and the maturity of relational database technology, it may not pay off to re-design another physical model for storing symbolic data. QAGen opts to leverage existing relational databases to implement the concept of a symbolic database. To that end, a natural idea for storing symbolic data is to store the data in columns of tables, introduce a user-defined type (UDT) to describe the columns, and use SQL user-defined functions to implement the symbolic operations. However, symbolic operations (e.g., a join that controls the output size and distribution) are too complex to be implemented by SQL user-defined functions. As a result, we propose to store symbols (and associated predicates) in relational databases by simply using the `varchar` SQL data type and letting the QAGen symbolic query engine operate on a relational database directly. This way, we integrate the power of various access methods brought by the relational database engine into SQP.

The next interesting question is how to normalize a symbolic relation for efficient SQP. From the definition of a symbol, we know that a symbol may be associated with a set of predicates. For example, symbol $\$a1$ may have a predicate $[\$a1 \geq p_1]$ associated with it. As we will see later, many symbolic executions of operators impose some predicates (from now on, we use the term predicate instead of constraint) on the symbols. Therefore, a symbol may be associated with many predicates. As a result, QAGen stores the predicates of a symbol in a separate relational table called PTable. Reusing Fig. 2 again, symbolic relation R can be represented by a normal table in a RDBMS named R with the schema: R(a: *varchar*, b: *varchar*) and a table named PTable with the schema: PTable(symbol: *varchar*, predicate: *varchar*). After a simple selection $\sigma_{a \geq p_1} R$ on table R, the relational representation of symbolic table R is:

a	b
$\$a1$	$\$b1$
$\$a2$	$\$b2$

Table R (2 tuples)

symbol	predicate
$\$a1$	$[\$a1 \geq p_1]$
$\$a2$	$[\$a2 < p_1]$

PTable (2 tuples)

Finally, note that even if a symbol $\$a1$ is representing a concrete value (e.g., symbol $\$a1$ has a value 5 after a selection $\sigma_{a=5} R$), the concrete value of $\$a1$ is still expressed as a concrete predicate $[\$a1 = 5]$ in the PTable.

4.2 Symbolic query evaluation

The major difference between SQP and traditional query processing is that the input (and thus the output) of each operator is symbolic data. The flexibility of symbolic data allows an operator to control its internal operation and thus its output. As in traditional query processing, an operator in SQP is implemented as an iterator. Therefore, the interface of an operator is the same as in traditional query processing which consists of three methods: open(), getNext() and close().

Next, we present the knobs and the algorithms for each operator through a running example. Unless stated otherwise, the following sub-sections only show the details of the getNext() method of each operator. All other aspects (e.g., open() and close()) are straightforward so that they may be omitted for brevity. The running example is a three-way join query which demonstrates the details of the symbolic execution of selection, equi-join, aggregation, and projection. We also discuss some special cases of these operators. Figure 5a

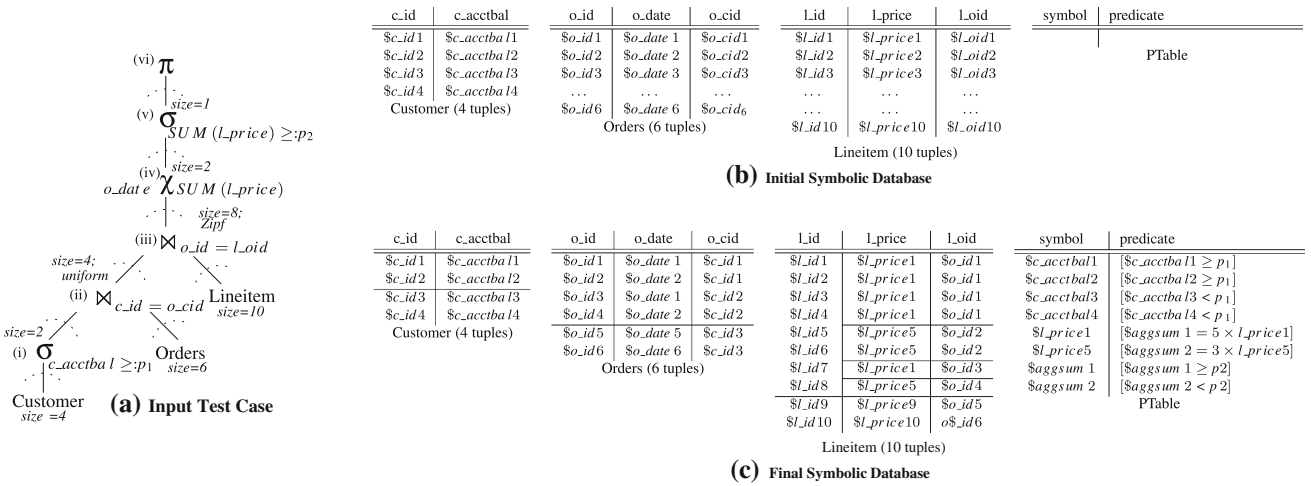

Fig. 5 Running example

shows the input query tree (with all knobs and their values given). The example is based on the following simplified TPC-H schema (primary keys are underlined):

Customer (c_id *int*, c_acctbal *float*)
Orders (o_id *int*, o_date *date*, o_cid `REFERENCE` Customer)
Lineitem (l_id *int*, l_price *float*, l_oid `REFERENCE` Orders)

4.2.1 Symbolic execution of the table operator

Knob:	Table Size (compulsory)

In QAGen, a base table in a query tree is regarded as an operator. During the open() method, it creates a relational table in a RDBMS with the attributes specified on input schema M. According to the designed storage model, all attributes are in the SQL data type `varchar`. Next, it fills up the table by creating new symbolic tuples until it reaches the defined table size. Each symbol in the newly created tuples is named using the attribute name as prefix and a unique identification number. Therefore, at the beginning of SQP, each symbol in the base table should be unique. Figure 5b shows the relational representation of the three symbolic relations Customer, Orders and Lineitem for the running example. The getNext() method of the table operator is the same as the traditional Table-Scan operator that returns a tuple to its parent or returns null (an end-of-result message) if all tuples have been returned. Note that if the same table is used multiple times in the query, then the table operator only creates and fills the base symbolic table once.

Primary keys, unique and not null constraints are already enforced because all symbols are initially unique. Foreign key constraints related to the query are taken care of by the join operator directly.

4.2.2 Symbolic execution of the selection operator

Knob:	Output Cardinality c (optional; default value = input size)

Let I be the input and O be the output of the selection operator σ and let p be the selection predicate. The symbolic execution of the selection operator controls the cardinality c of the output. Depending on the input characteristics, the difficulty and the algorithms for this execution are completely different. Generally, there are two different cases.

Case 1: Input is not pre-grouped w.r.t. the selection attribute(s) This is case (a) in Fig. 4 and the selections in the running example (Fig. 5a operator (i) and (v)) are in this case. This implementation is chosen by the query analyzer when the input is not pre-grouped w.r.t. the selection attribute(s) and it is the usual case for most queries. In this case, the selection operator controls the output as follows:

1. During its getNext() method, read in a tuple t by invoking getNext() on its child operator and process with [Positive Tuple Annotation] if the output cardinality has not reached c. Else proceed to [Negative Tuple Post Processing] and then return null to its parent.
2. [Positive Tuple Processing] If the output cardinality has not reached c, then (a) for each symbol s in t that participates in the selection predicate p, insert a corresponding tuple $\langle s, p\rangle$ to the PTable; and (b) return this tuple t to its parent.
3. [Negative Tuple Post Processing] However, if the output cardinality has reached c, then fetch all the remaining tuples I^- from input I. For each symbol s of tuple t in I^- that participates in the selection predicate p, insert a corresponding tuple $\langle s, \neg p\rangle$ to the PTable, and repeat this step until calling getNext() on its child has no more tuples (returns null).

Each getNext() call on the selection operator returns to its parent a *positive* tuple that satisfies the selection predicate p until the output cardinality has been reached. Moreover, to ensure that all negative tuples (i.e., tuples obtained from the child operator after the output cardinality has been reached) would not get some instantiated values later in the data instantiation phase that ends up passing the selection predicate, the selection operator associates the negation of predicate p with those negative tuples. In the running example, attribute $c\_acctbal$ in the selection predicate $[c\_acctbal \geq p_1]$ of operator (i) is not pre-grouped, because the data comes directly from the base Customer table. Since the output cardinality c of the selection operator is 2, the selection operator associates the positive predicate $[c\_acctbal \geq p_1]$ to symbols $\$c\_acctbal1$ and $\$c\_acctbal2$ of the first two input tuples and associates the negated predicate $[c\_acctbal < p_1]$ to symbols $\$c\_acctbal3$ and $\$c\_acctbal4$ of the rest of the input tuples. Table 1(i) shows the output of the selection operator and Table 1(ii) shows the content of the PTable after the selection.

Case 2: Input is pre-grouped w.r.t. the selection attribute(s)

This is case (b) in Fig. 4. This implementation is chosen by the query analyzer when the input is pre-grouped with respect to any attribute that appears in the selection predicate p. In this case, we can show that the problem of controlling the output cardinality is reducible to the subset-sum problem.

The subset-sum problem [17] takes as input an integer sum c and a set of integers $C = \{c_1, c_2, \ldots, c_m\}$, and outputs whether there exists a subset $C^+ \subseteq C$ such that $\sum_{c_i \in C^+} c_i = c$. Consider Fig. 6, which is an example of pre-grouped input

Table 1 After selection

c_id	c_acctbal
(i) Output of σ; 2 tuples	
$\$c\_id1$	$\$c\_acctbal1$
$\$c\_id2$	$\$c\_acctbal2$
Symbol	**Predicate**
(ii) PTable	
$\$c\_acctbal1$	$[\$c\_acctbal1 \geq p_1]$
$\$c\_acctbal2$	$[\$c\_acctbal2 \geq p_1]$
$\$c\_acctbal3$	$[\$c\_acctbal3 < p_1]$
$\$c\_acctbal4$	$[\$c\_acctbal4 < p_1]$

k	
$\$k1$	} *e.g.* $c_1 = 5$ *times*
$\$k2$	} *e.g.* $c_2 = 4$ *times*
$\$k3$	} *e.g.* $c_3 = 3$ *times*
$\$k4$	} *e.g.* $c_4 = 1$ *times*
...	
$\$km$	} c_m *times*
	Input I

Fig. 6 Pre-grouped selection

of a selection. Input I defines one attribute k and has in total $\sum c_i$ rows. The rows in I are clustered in m groups, where the ith group has c_i tuples with the same symbolic value k_i ($i \leq m$). We now search for a subset of those m groups in I such that the output has the size c. Assume, we find such a subset, i.e., the symbolic values of those groups which result in the output with size c. The groups returned by such a search induce a solution for the original subset-sum problem.

The subset-sum problem is a weakly $\mathcal{NP}$-complete problem and there exists a pseudopolynomial algorithm which uses dynamic programming to solve it [17]. The complexity of the dynamic programming algorithm is $\mathcal{O}(cm)$, where c is the expected output cardinality and m is the number of different groups in I. When c is large, the dynamic programming algorithm runs very slow. Furthermore, it is also possible that there is no subset in the input whose sum meets c as well. As a result, when the query analyzer detects that the input of a selection is pre-grouped, it allows the user to specify the following knob in addition to the output cardinality knob:

Knob:	Approximation ratio ϵ

The approximation ratio knob allows the selection to return an approximate number of tuples rather than the exact number of tuples that is specified by the testers and this is a new feature of this version of QAGen.

There are several approximation schemes in the literature to solve the subset-sum problem (e.g., [22,24,33]). However, these approximation schemes are not directly applicable in our case. We illustrate this problem using a toy test case

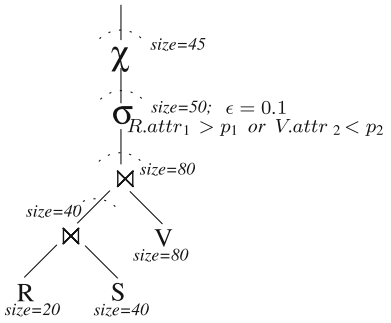

Fig. 7 A test case with the approximation ratio knob

(see Fig. 7). The test query in the test case is a two-way join query with an aggregation. In Fig. 7, the tester defines that the output cardinality of the selection as 50 tuples with an approximation ratio of 0.1. Assume that the input of the selection in Fig. 7 has 80 tuples but they are pre-grouped into three clusters (a cluster c_1 consists of 36 tuples, and two clusters c_2 and c_3 consist of 22 tuples each) with respect to both attributes $attr_1$ and $attr_2$ after the two-way join. In order to pick the correct subset of pre-grouped tuples with a total cardinality of c ($c = 50$ in the example), the selection operator needs to solve the subset-sum problem by an approximation scheme. Unfortunately, all existing approximation schemes are designed to return a subset whose sum is *smaller than* (or equal to) the target sum. Consequently, it is possible that an approximation scheme suggests picking clusters c_2 and c_3 from the pre-grouped input, such that the selection returns a total of 44 tuples (which is actually the best solution with target sum as 50 tuples) as output. However, if the selection really returns 44 tuples, then the upper aggregation operator χ in Fig. 7 would experience a "lack-of-tuple" error (it expects to have 45 or more input tuples). Even though the target users of QAGen are experienced testers, it is still difficult for them to specify a semantically correct test case when the system allows tolerances on the operator's cardinality constraint. This practical problem drove us to develop an approximation scheme that returns a subset with sum *greater than or equal to* the target sum c and has an approximation ratio ϵ. We call this new problem as the *Overweight Subset-Sum Problem* and it requires non-trivial modifications to the current approximation schemes. Note that an alternative method to solving the "lack-of-tuple" error is to consider the information from the parent operator. For example, in Fig. 7, by considering the cardinality requirement of the aggregation operator, the selection operator can look for a subset of clusters whose totally cardinality in the range from 45 to 50 tuples. Then, the problem can be reduced to a subset range-sum problem. In this work, we opt to solve the problem as an overweight subset-sum problem because it allows a user to control the approximation ratio.

Algorithm APPROXIMATE_OVERWEIGHT_SUBSET_SUM(P)
Input: (a) A list of sorted integers $C = [c_1, c_2, ..., c_m]$ where $c_i < c_{i+1}$ (b) Target sum c, (c) Approximation ratio ϵ
Output: A subset of integers $C^+ \subseteq C$ such that $c \leq \sum_{c_i \in C^+} c_i$ with approximation ratio ϵ
1. **if** $\exists c_i \in C, c_i \geq c$
2. **then** Trim C by removing elements $c_{i+1}, ..., c_m$
3. Set the largest possible optimal solution p as $p = c_1 + c_2 + ... + c_r \geq c$ where $c_1 + c_2 + ... + c_{r-1} < c$. If $c_r \geq c$, **return** $\{c_r\}$. If no such r exists, **return** "no solution exists".
4. Set quantization factor $d = (\epsilon/2)^2 p$
5. Set number of buckets $g = \lceil p/d \rceil + \min\{r, \lceil 2/\epsilon \rceil\}$
6. Initialize $g+1$ approximate answer buckets $\mathcal{B} = \{B_0, B_1, ..., B_g\}$
7. Initialize a subset-sum array X of size $g+1$ where $X[i]$ stores the subset-sum of the elements in B_i. Set $X[0] = 0$ and $X[i] = -1$ $(1 \leq i \leq g)$
8. Set list $S = [c_1, c_2, ..., c_u]$ where $c_u < (\epsilon/2)p$
9. Set list $L = [c_{u+1}, c_{u+2}, ..., c_m]$ where $c_{u+1} \geq (\epsilon/2)p$
10. Return S as the answer if L is empty
11. **for** each number $c_i \in L$
12. Set the quantized value of v_i of c_i as $\lceil c_i/d \rceil$
13. **for** each $j = g - v_i$ down-to 0
14. **if** $X[j] \neq -1$
15. **then if** $X[j + v_i] < X[j] + c_i$
16. **then** set $B_{j+v_i} = B_j \cup \{c_i\}$,
17. set $X[j + v_i] = X[j] + c_i$
18. **for** each bucket $B_i \in \mathcal{B}$ with $X[i] \neq -1$
19. set $j = 0$
20. **while** $X[i] < c$
21. set $B_i = B_i \cup \{c_j\}$, where c_j is the j-number in list S,
22. set $X[i] = X[i] + c_j$
23. $j = j + 1$
24. **return** B_i, where $X[i] = min(X[j])$ for all $0 \leq j \leq g$ and $X[j] \geq c$;

Fig. 8 Approximation scheme for the overweight subset-sum problem

Our new approximation scheme is based on the "quantization method" [22] and consists of two phases. It takes a list C of sorted numbers as input. Then, it first separates the input list of numbers into two lists: large number list L and small number list S. In the first phase, it tries to quickly come up with a set of approximation solutions by only considering the numbers with large values (i.e., only elements in L). Then, in the second phase, it tries to fine tune the approximation solutions by the set of small numbers in S.

Figure 8 shows the pseudocode of the approximation scheme. In the beginning, it trims input list C if it contains more than one number which has a value greater than or equal to the target sum c. For example, assume input list C is [1, 2, 5, 6, 13, 27, 44, 47, 48], the target sum c is 30, and the approximation ratio ϵ is 0.1. After line (1–2), C becomes [1, 2, 5, 6, 13, 27, 44] because 47 and 48 cannot be part of the answer. Then, it tries to quantize the large values into different buckets (line 4–7) in order to minimize the number of subsequent operations from line 11 to line 24. Based on the quantization factor d, the algorithm quantizes the input list of numbers into g buckets. The quantization factor d is carefully chosen such that it is large enough to give a manageable number of buckets and at the same time respecting the error bound given by the approximation ratio ϵ [22]. The quantization factor d is computed based on the approximation ratio ϵ and one of the possible subset-sums p. Such a p value is found (line 3) by adding $c_1, c_2, \ldots$ until the sum is at least the target sum c; if no such value is found, the sum of all values in C must be less than c, and we can conclude that there is no solution for the overweight subset-sum problem. An interesting special case is that, if the last value of the sum, c_r, is at least c, we immediately know $\{c_r\}$ is the desired *optimal* solution to the overweight subset-sum problem. X is a subset-sum array. Entry $X[i]$ stores the subset-sum of the elements in bucket B_i (line 7). Initially, $X[0]$ is set to 0 as a boundary condition and $X[i]$ (where $i \neq 0$) is set to -1 to make sure a subset-sum cannot exceed $i \times d$ in any case.

In the example, $p = 1 + 2 + 5 + 6 + 13 + 27 = 54$, and thus the quantization level d and the number of buckets g are 0.135 and 406, respectively. Afterwards, the algorithm creates $g + 1$ approximate answer buckets $\mathcal{B}$ and a subset-sum array X, where each approximate answer bucket B_i will hold a set of numbers whose the sum is close to a factor i of the quantization factor d (i.e., the subset-sum is close to $i \times d$) and $X[i]$ represents the total sum of numbers in B_i.

As mentioned, the input list of numbers is separated into two lists S and L according to the numbers' value (lines 8–9). In the example, the small list S consists of the first two numbers 1 and 2 in the input list C and the large list L consists of all the rest of the numbers [5, 6, 13, 27, 44]. Then, the first phase (lines 11–17) begins by examining each number in the large number list L and tries to assign the number into different buckets. For example, the first number in L is 5 and its quantized values is $\lceil 5/0.135 \rceil = 38$. Therefore, the algorithm sets $B_{38} = \{5\}$ and the corresponding subset-sum array entry $X[38]$ has a value of 5. Similarly, for the second number 6 in L, its quantized value is $\lceil 6/0.135 \rceil = 44$. As a result, the algorithm sets B_{44} to be $\{6\}$, updates $X[44]$ to be 6, sets B_{82} to be $\{5, 6\}$ and updates $X[82]$ to have a value of 11 $(= 5 + 6)$. If a bucket is non-empty, the algorithm only updates the bucket (and its corresponding subset-sum in X) if the updated subset-sum is larger than the current subset-sum of that bucket (lines 15–17).

In the second phase (lines 18–23), the algorithm tries to fine tune each approximate answer bucket B by adding the numbers in the small list S, one-by-one, until it exceeds the target sum c. Afterwards, the algorithm scans array X and identifies the subset which has the smallest subset-sum that is greater than the target sum c. Finally, it returns the corresponding subset in B as the final result.

The complexity of our proposed approximation scheme is $\mathcal{O}(m/\epsilon^2)$. For the correctness proof and the complexity analysis, we refer the readers to [28]. We now reuse Fig. 6 to illustrate the overall algorithm of the selection operator. Assume that the input has 13 tuples which are clustered into 4

groups with symbol $\$k1, \$k2, \$k3$, and $\$k4$, respectively. Furthermore, assume that the output cardinality and the approximation ratio is defined as 7 tuples and 0.2, respectively. The pre-grouped input selection controls the output as follows:

1. [Subset-sum solving] During its open() method, (a) materialize input I of the selection operator; (b) extract the pre-group size (e.g., $c_1 = 5, c_2 = 4, c_3 = 3, c_4 = 1$) of each symbol k_i by executing "*Select Count(k) From I Group By k Order By Count(k)*" on the materialized input; (c) invoke the approximation scheme in Fig. 8 with the pre-group sizes (the set of numbers), the output cardinality (the target sum), and the approximation ratio ϵ as input. The output of this step is a subset of symbols K^+ in I such that the output cardinality (approximately) meets the constraint (e.g., $K^+ = \{\$k1, \$k3\}$ because $c_1 + c_3 = 5 + 3 = 8 \geq c$). If no such subset exists, then stop processing and report this error to the user.
2. [Positive Tuple Processing] During getNext(), (a) for each symbol k_i in K^+, read all tuples I^+ from the materialized input of I which have k_i as the value of attribute k; (b) for each symbol s that participates in the selection predicate p in tuple t of I^+, insert a corresponding tuple $\langle s, p \rangle$ to the PTable; (c) return tuple t to the parent.
3. [Negative Tuple Post Processing] This step is the same as the Negative Tuple Post Processing step in the simple case (Sect. 4.2.2 case 1) that annotates negative predicates to each negative tuple.

Note that, in this case, the selection is a blocking operation because it needs to read all the tuples from input I first in order to solve the subset-sum problem. One optimization for this case is that if c is equal to the input size of I, then all input tuples must be returned to its parent and thus the subset-sum solving function can be skipped even though the input data is pre-grouped.

4.2.3 Symbolic execution of the equi-join operator

Knob:	Output Cardinality c (optional; default value = size of the non-distinct input)

Let R and S be the inputs, O be the output, and p be the simple equality predicate $j = k$ where R is the not-pre-grouped w.r.t. join attribute j, and k is the join attribute on S that refers to j by a foreign key relationship. The symbolic execution of the equi-join operator ensures that the join result size is c. Again, depending on whether the input is pre-grouped or not, the solutions are different.

Case 1: Input is not pre-grouped w.r.t. join attribute k.

This is case (c) in Fig. 4, where input S is not pre-grouped w.r.t. join attribute k. In this case, it is possible to support one more knob on the equi-join operation:

Knob:	Join Key Distribution b (optional; choices = [Uniform or Zipf]; default = Uniform)

The join key distribution b defines how many tuples of input S join with each individual tuple in input R. For example, if the join key distribution is uniform, then each tuple in R joins with roughly the same number of tuples in S. Both join operators in Fig. 5a fall into this case. In this case, the equi-join operator (which supports both output cardinality c and distribution b) controls the output as follows:

1. [Distribution instantiating] During its open() method, instantiate a distribution generator Z, with the size of R as domain (denoted by n), the output cardinality c as frequency, and the distribution type b as input. This distribution generator Z can be the one that has been proposed earlier (e.g., [6,19]) or any statistical packages that generate n numbers $m_1, m_2, \ldots, m_n$ following Uniform or Zipf [37] distribution with a total frequency of c.[4] The distribution generator Z is an iterator with a getNext() method. For the ith call on the getNext() method ($0 < i \leq n$), it returns the expected frequency m_i of the ith number under distribution b.
2. During its getNext() call, if the output cardinality has not yet reached c, then (a) check if $m_i = 0$ or if m_i has not yet initialized, and, if so, initialize m_i by calling getNext() on Z and get a tuple r^+ from R (m_i is the total number of tuples from S that should join with r^+); (b) get a tuple s^+ from S and decrease m_i by one; (c) join tuple r^+ with s^+ according to [Positive Tuple Joining] below; (d) return the joined tuple to the parent. However, during the getNext() call, if the output cardinality has reached c already, then process [Negative Tuple Joining] below, and return null to its parent.
3. [Positive Tuple Joining] If the output cardinality has not reached c, then (a) for tuple s^+, replace symbol $s^+.k$, which is the symbol of the join key attribute k of tuple s^+, by symbol $r^+.j$, which is the symbol of the join key attribute j of tuple r^+. After this, tuple r^+ and tuple s^+ should share exactly the same symbol on their join attributes. Note that the replacement of symbols in this step is done on both tuples loaded in the memory and the related

[4] The technique is also applicable to other distributions as long as the corresponding distribution generator is available. Furthermore, although rarely happens, it is possible that the n numbers returned by the distribution generator may sum up to be larger than c. In this case, as users of QAGen are prepared to getting approximation answers in some cases, the process continues but the users will get informed after the data generation.

Table 2 After joining

c_acctbal	o_id	o_date	**c_id = o_cid**
(i) Output of $(\sigma(Customer) \bowtie Order)$; 4 tuples			
$\$c\_acctbal1$	$\$o\_id1$	$\$o\_date1$	**$c_id1**
$\$c\_acctbal1$	$\$o\_id2$	$\$o\_date2$	**$c_id1**
$\$c\_acctbal2$	$\$o\_id3$	$\$o\_date3$	**$c_id2**
$\$c\_acctbal2$	$\$o\_id4$	$\$o\_date4$	**$c_id2**
o_id	o_date	**o_cid**	
(ii) Orders (4 pos, 2 neg)			
$\$o\_id1$	$\$o\_date1$	**$c_id1**	
$\$o\_id2$	$\$o\_date2$	**$c_id1**	
$\$o\_id3$	$\$o\_date3$	**$c_id2**	
$\$o\_id4$	$\$o\_date4$	**$c_id2**	
$\$o\_id5$	$\$o\_date5$	***$c_id3***	
$\$o\_id6$	$\$o\_date6$	***$c_id4***	

tuples in base table as well (using an SQL statement like "*Update k.BaseTable Set k=r^+.j WHERE k=s^+.k*" to update the symbols on the base table where join attribute k comes from); (b) perform an equi-join on tuple r^+ and s^+.

4. [Negative Tuple Joining] However, if the output cardinality has reached c, then fetch all the remaining tuples S^- from input S. For each tuple s^- in S^-, randomly look up a symbol j^- on the join key j in the set minus between the base table where join attribute j originates from and R (that step can be implemented by composing an SQL statement using the SQL `MINUS` operator), replace $s^-.k$ with symbol j^-. This replacement is done on the base tables only because these tuples are not returned to the parent.

In the running example (Fig. 5), after the selection on table Customer (operator (i)), the next operator is a join between the selection output (Table A(i) in Sect. 4.2.2) and table Orders. The output cardinality c of that join (operator (ii)) is 4 and the join key distribution is uniform. Since the input of the join is not pre-grouped w.r.t. the join key o_cid, the query analyzer uses the algorithm above to perform the equi-join. First, the distribution generator Z generates 2 numbers (which is the size of input R), with total frequency of 4 (output cardinality), and uniform distribution. Assume Z returns the sequence {2, 2}. This means that the first customer $\$c\_id1$ should take 2 orders ($\$o\_id1$ and $\$o\_id2$) and the second customer $\$c\_id2$ should also take 2 orders ($\$o\_id3$ and $\$o\_id4$). As a result, symbols $\$o\_cid1$ and $\$o\_cid2$ from the Orders table should be replaced by $\$c\_id1$ and symbols $\$o\_cid3$ and $\$o\_cid4$ from the Orders table should be replaced by $\$c\_id2$ (Step 3 above). In order to fulfill the foreign key constraint on those tuples which do not join, Step 4 (Negative Tuple Joining) replaces o_cid5 and o_cid6 by customers that did not pass through the selection filter (i.e., customer $\$c\_id3$ and $\$c\_id4$) randomly. Table 2(i) below shows the output of the join and Table 2(ii) shows the updated Orders table (updated join keys are **bold**).

After the join operation above, the next operator in the running example is another join between the above join results (Table 2(i)) and the base Lineitem table (Fig. 5b(iii)). Again, the input of the join on the join key l_oid of the Lineitem table is not pre-grouped and thus the above equi-join algorithm is chosen by the query analyzer. Assume that the distribution generator generates a Zipf sequence {4,2,1,1} for the four tuples in Table 2(i) to join with 8 out of 10 line-items (where 8 is the user-specified output cardinality of this join operation). Therefore, it produces the following output (updated join keys are **bold**):

Finally, note that if the two inputs of an equi-join are base tables (with foreign key constraint), then the output cardinality knob is disabled by the query analyzer. This is because in that case, all tuples from input S must join with a tuple from input R and thus the output cardinality must be same as the size of S.

Case 2: Input is pre-grouped w.r.t. join attribute k.

This is case (d) in Fig. 4 and this implementation is chosen by the query analyzer when input S is pre-grouped w.r.t. join attribute k. This sometimes happens when a preceding join introduces a distribution on k as in the example in Fig. 2. In the following we show that if the input is pre-grouped w.r.t. join attribute k of an equi-join, then the problem of controlling the output cardinality (even without the join key distribution) is also reducible to the subset-sum problem.

Consider tables R and S in Fig. 9, which are the inputs of such a join. Table R has one attribute j with l tuples all using distinct symbolic values j_i ($i \leq l$). Table S also defines only one attribute k and has in total $\sum c_i$ rows. The rows in S are

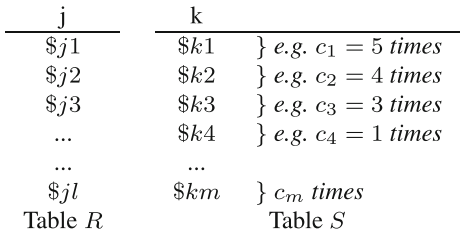

Fig. 9 Pre-grouped equi-join

clustered into m groups, where the ith group has exactly c_i tuples using the same symbolic value k_i ($i \leq m$). We now search for a subset of those m groups in S that join with arbitrary tuples in R so that the output has size c. Assume that we find such a subset, i.e., the symbolic values of those groups which result in the output with size c. The groups returned by such a search induce a solution for the original subset-sum problem.

For testing the feature of a DBMS, again, it is sufficient for the equi-join to return an approximate number of tuples that is close to the user specified cardinality. As a result, when the query analyzer detects that one of the equi-join inputs is pre-grouped, then it allows the user to specify the following knob in addition to the output cardinality knob:

Knob:	Approximation Ratio ϵ

Again, this is a blocking operator because it needs to read all the input tuples from S first (to solve the subset-sum problem). Similar to the optimization in the selection operator, if c is equal to the input size of S, then all tuples of S must be joined with R and the subset-sum solving function can be skipped even though the data is pre-grouped.

We reuse Fig. 9 to illustrate the algorithm. Assume the join is on Table R and Table S and the join predicate is $j = k$. Assume Table R has three tuples ($\langle \$j1 \rangle$, $\langle \$j2 \rangle$, $\langle \$j3 \rangle$), and Table S has 12 tuples which are clustered into 4 groups with symbols $\$k1$, $\$k2$, $\$k3$, $\$k4$, respectively. Furthermore, assume the join on R and S is specified with an output cardinality as $c = 7$. The pre-grouped input equi-join controls the output as follows:

1. [Subset-sum solving] During its open() method, (a) materialize input S of the join operator; (b) extract the pre-group size (e.g. $c_1 = 5$, $c_2 = 4$, $c_3 = 3$, $c_4 = 1$) of each symbol k_i by executing "*Select Count(k) From S Group By k Order By Count(k) Desc*" on the materialized input; (c) invoke the approximation scheme in Fig. 8 with the pre-group sizes (the set of numbers), the output cardinality (the target sum), and the approximation ratio ϵ as input. The output of this step is a subset of symbols K^+ in I such that the output cardinality (approximately) meets the constraint (e.g., $K^+ = \{\$k1, \$k3\}$ because $c_1 + c_3 = 5 + 3 = 8 \geq c$). If no such subset exists, then stop processing and report this error to the user.
2. [Positive Tuple Joining] During getNext(), (a) for each symbol k_i in K^+, read all tuples S^+ from the materialized input of S which have k_i as the value of attribute k; (b) afterwards, call getNext() on R once and get a tuple r, join all tuples in S^+ with r by replacing the join key symbols in S^+ with the join key symbols in r. For example, the first five $\$k1$ symbols in S are replaced with $\$j1$ and the three $\$k3$ symbols in S are replaced with $\$j2$ (again, these replacements are done on symbols loaded in the memory and the changes are propagated to the base tables where j and k originate from); (c) return the joined tuples to the parent.
3. [Negative Tuple Joining] This step is the same as the Negative Tuple Joining step in the simple case (Sect. 4.2.3 case 1) that joins the negative tuples in input R with the negative tuples in input S.

4.2.4 Symbolic execution of the aggregation operator

Knob:	Output Cardinality c (optional; default value = input size)

Let I be the input and O be the output of the aggregation operator and f be the aggregation function. The symbolic execution of the aggregation operator controls the size of the output as c.

Simple Aggregation This is the simplest case of aggregation where there is no grouping operation (i.e,. no GROUP-BY keyword) defined on the query. In this case, the query analyzer disables the output cardinality knob because the output cardinality always equals to one. In SQL, there are five aggregation functions: SUM, MIN, MAX, AVG, COUNT. For simple aggregation, the solutions are very similar for both pre-grouped or non-pre-grouped input on the attribute(s) in f. The following shows the case of non-pre-grouped input:

Let $expr$ be the expression in the aggregation function f which consists of at least a non-empty set of symbols S in $expr$ and let the size of input I be n.

1. SUM($expr$). During its getNext() method, (a) the aggregation operator consumes all n tuples from I; (b) for each symbol s in S, adds a tuple $\langle s, [aggsum = expr_1 + expr_2 + \ldots + expr_n] \rangle$ to the PTable, where $expr_i$ is the corresponding expression on the ith input tuple; and (c) returns symbolic tuple $\langle \$aggsum \rangle$ as output. As an example, assume there is an aggregation function SUM(l_price) on top of the join result in Table C(i) of the previous section. Then, this operator returns one tuple $\langle \$aggsum \rangle$ to its parent and adds 8 tuples (e.g., the 2nd

inserted tuple is $\langle \$l\_price2, [\$aggsum = \$l\_price1 + \$l\_price2 + \ldots + \$l\_price8] \rangle$) to the PTable.
The above is a base case. If there are no additional constraints that will be further imposed on the predicate symbols, the aggregation operator will optimize the number and the size of the above predicates by inserting only one tuple $\langle \$l\_price1, [\$aggsum = \$l\_price1 \times 8] \rangle$ to the PTable and replacing symbols $\$l\_price2, \ldots, \$l\_price8$ by symbol $\$l\_price1$ on the base table. One reason for doing this is the size of the input may be very big. If that is the case, the extremely long predicate may exceed the SQL `varchar` size upper bound. Another reason is to insert fewer tuples in the PTable. However, the most important reason is that the cost of a constraint solver call is exponential to the size of the input formula in the worst case. Therefore, this optimization reduces the time of the later data instantiation phase. However, there is a trade-off: for each input tuple, the operator has to update the corresponding symbol in the base table where this symbol originates from.

2. MIN($expr$). If possible, the MIN aggregation operator also uses similar predicate optimization as SUM aggregation. During its getNext() method, it (a) regards the first expression $expr_1$ as the minimum value and returns $\langle expr_1 \rangle$ as output; and (b) replaces the expression $expr_i$ in the remaining tuples (where $2 < i \leq n$) by the second expression $expr_2$ and inserts two tuples $\langle expr_1, [expr_1 < expr_2] \rangle$ and $\langle expr_2, [expr_1 < expr_2] \rangle$ to the PTable. Note that this optimization must be aware of whether the input is pre-grouped or not. If it is, not only the first but all tuples with $expr_1$ are kept and the remaining are replaced with symbol $expr_2$.
As an example, assume that there is an aggregation function MIN(l_price) on top of the join result in Table C(i). Then, this operator returns $\langle \$l\_price1 \rangle$ as output and inserts two tuples into the PTable: $\langle \$l\_price1, [\$l\_price1 < \$l\_price2] \rangle$ and $\langle \$l\_price2, [\$l\_price1 < \$l\_price2] \rangle$ to the PTable. Moreover, according to step (b) above, $\$l\_price3$, $\$l\_price4$, …, $\$l\_price8$ are replaced by $\$l\_price2$ on the base table.
3. MAX($expr$). During its getNext() method, it (a) regards the first expression $expr_1$ as the maximum value and returns $\langle expr_1 \rangle$ as output; and (b) replaces the expression $expr_i$ in the remaining tuples (where $2 < i \leq n$) by the second expression $expr_2$ and inserts two tuples $\langle expr_1, [expr_1 > expr_2] \rangle$ and $\langle expr_2, [expr_1 > expr_2] \rangle$ to the PTable.
4. COUNT($expr$). The aggregation operator handles the COUNT aggregation function in a similar way to traditional query processing. During its getNext() method, (a) it counts the number of input tuples, n; (b) add a tuple $\langle \$aggcount, \$aggcount = n \rangle$ to the PTable; and (c) returns a symbolic tuple $\langle \$aggcount \rangle$ as output.
5. AVG($expr$). It is the similar to the case of the SUM aggregation. During its getNext() method, (a) the aggregation operator consumes all n tuples from I; (b) for each symbol s in S, it adds a tuple $\langle s, [\$aggavg = (expr_1 + expr_2 + \ldots + expr_n)/n] \rangle$ to the PTable, where $expr_i$ is the corresponding expression on the ith input tuple; and (c) returns symbolic tuple $\langle aggavg \rangle$ as output. The optimization can be illustrated by our example: It adds only one tuple $\langle \$l\_price1, [\$aggavg = \$l\_price1] \rangle$ to the PTable and replaces symbols $\$l\_price2, \ldots, \$l\_price8$ by symbol $\$l\_price1$ on the base table.

In general, combinations of different aggregation functions in one operator (e.g. MIN($expr1$) + MAX($expr2$)) need different yet similar solutions. Their solutions are straightforward and we do not cover them here.

Single GROUP-BY Attribute When the aggregation operator has one GROUP-BY attribute, the output cardinality c defines how to assign the input tuples into c output groups. Let g be the single grouping attribute. For all algorithms we assume that g has no unique constraint in the database schema. Otherwise, the grouping is predefined by the input already and the query analyzer disables all knobs on the aggregation operator for the user. Again, this symbolic operation of aggregation can be divided into two cases:

Case 1: Input is not pre-grouped w.r.t. the grouping attribute

In addition to the cardinality knob, when the symbols of the grouping attribute g in the input are not pre-grouped, it is possible to support one more knob:

Knob:	Group Distribution b (optional; choices = [Uniform or Zipf]; default = Uniform)

The group distribution b defines how to distribute the input tuples into c predefined output groups. In this case, the aggregation operator controls the output as follows:

1. [Distribution instantiating] During its open() method, instantiate a distribution generator Z, with the size of I (denoted by n) as frequency, the output cardinality c as domain, and the distribution type b as input. The distribution generator is the same one as that for doing equi-join (Sect. 4.2.3). It generates c numbers $m_1, m_2, \ldots, m_c$, and the ith call on its getNext() method ($0 < i \leq c$) returns the expected frequency m_i of the ith number under distribution b.
2. During getNext(), call Z.getNext() to get a frequency m_i, fetch m_i tuples (let them be I_i) from I and execute the following steps. If there are no more tuples from its child operator, return null to the parent.
3. [Group assigning] For each tuple t in I_i, except the first tuple t' in I_i, replace symbol $t.g$, which is the symbol of

Table 3 After 2-way join

c_id	c_acctbal	o_date	o_cid	l_id	l_price	**o_id = l_oid**
(i) Output of $(\sigma(Customer) \bowtie Order) \bowtie Lineitem$. 8 tuples						
$\$c\_id1$	$\$c\_acctbal1$	$\$o\_date1$	$\$o\_cid1$	$\$l\_id1$	$\$l\_price1$	**$o_id1**
$\$c\_id1$	$\$c\_acctbal1$	$\$o\_date1$	$\$o\_cid1$	$\$l\_id2$	$\$l\_price2$	**$o_id1**
$\$c\_id1$	$\$c\_acctbal1$	$\$o\_date1$	$\$o\_cid1$	$\$l\_id3$	$\$l\_price3$	**$o_id1**
$\$c\_id1$	$\$c\_acctbal1$	$\$o\_date1$	$\$o\_cid1$	$\$l\_id4$	$\$l\_price4$	**$o_id1**
$\$c\_id1$	$\$c\_acctbal1$	$\$o\_date2$	$\$o\_cid1$	$\$l\_id5$	$\$l\_price5$	**$o_id2**
$\$c\_id1$	$\$c\_acctbal1$	$\$o\_date2$	$\$o\_cid1$	$\$l\_id6$	$\$l\_price6$	**$o_id2**
$\$c\_id2$	$\$c\_acctbal2$	$\$o\_date3$	$\$o\_cid2$	$\$l\_id7$	$\$l\_price7$	**$o_id3**
$\$c\_id2$	$\$c\_acctbal2$	$\$o\_date4$	$\$o\_cid2$	$\$l\_id8$	$\$l\_price8$	**$o_id4**
l_id	l_price	**l_oid**				
(ii) Lineitem (8 pos, 2 neg)						
$\$l\_id1$	$\$l\_price1$	**$o_id1**				
$\$l\_id2$	$\$l\_price2$	**$o_id1**				
$\$l\_id3$	$\$l\_price3$	**$o_id1**				
$\$l\_id4$	$\$l\_price4$	**$o_id1**				
$\$l\_id5$	$\$l\_price5$	**$o_id2**				
$\$l\_id6$	$\$l\_price6$	**$o_id2**				
$\$l\_id7$	$\$l\_price7$	**$o_id3**				
$\$l\_id8$	$\$l\_price8$	**$o_id4**				
$\$l\_id9$	$\$l\_price9$	***$o_id5***				
$\$l\_id10$	$\$l\_price10$	***$o_id6***				

the grouping attribute g of tuple t, by symbol $t'.g$. $t'.g$ is the symbol of the grouping attribute g of the first tuple t' in the ith group. Note that the replacement of symbols in this step is done on both the tuples loaded in the memory and the related tuples in the base table as well.

4. [Aggregating] Invoke the Simple Aggregation Operator mentioned early in this section with all the symbols participated in the aggregation function in I_i as input.
5. [Result Returning] Construct a new symbolic tuple $\langle t'.g, agg_i \rangle$, where agg_i is the symbolic tuple returned by the Simple Aggregation Operator for the ith group. Return the constructed tuple to its parent.

Sometimes, during the open() method, the distribution generator Z may return 0 when the distribution is skewed (e.g., Zipf distribution with high skew factor). In this case, it may happen that an output group does not get any input tuple and the final number of output groups may be fewer than the expected output cardinality. There are several ways to handle this case. One way is to regard this as an runtime error which lets users know that they should not specify such a highly skewed distribution when they ask for many output groups. Another way is to adjust the distribution generator Z such that it first assigns one tuple to each output group (which consumes c tuples), and then it starts assigning the remaining $n - c$ tuples according to the distribution generation algorithm. This ensures that the cardinality constraint is met. However, the final distribution may not strictly adhere to the original distribution. Here, we assume the user does not specify any contradicting constraints, therefore QAGen uses the first approach (i.e., return a runtime error).

Case 2: Input is pre-grouped w.r.t. the grouping attribute

When the input on the grouping attribute is pre-grouped, it is understandable that this operation does not support the group distribution knob as in the above case. But if the input is pre-grouped w.r.t. the grouping attribute and the output cardinality is the only specified knob, the operation is fairly simple.

The aggregation operator (iv) in the running example (Fig. 5a) falls into this case. Referring to Table 3(i), which is the input of the aggregation operator in the example. After several joins, the input is pre-grouped into four pre-groups w.r.t. o_date ($\$o\_date1 \times 4$; $\$o\_date2 \times 2$; $\$o\_date3 \times 1$; $\$o\_date4 \times 1$). In this case, the aggregation operator controls the output by assigning tuples from the same pre-group to the same output group and each pre-group is assigned into c output groups in a round-robin fashion. In the example, the output cardinality of the aggregation operator is two. The aggregation operator assigns the first pre-group (with $\$o\_date1$) which includes four tuples into the first

Table 4 After aggregation

o_date	SUM(l_price)
(i) Output of χ (2 tuples)	
$\$o\_date1$	$\$aggsum\_1$
$\$o\_date2$	$\$aggsum\_2$
Symbol	**Predicate**
(ii) PTable	
$\$c\_acctbal1$	$[\$c\_acctbal1 \geq p_1]$
$\$c\_acctbal2$	$[\$c\_acctbal2 \geq p_1]$
$\$c\_acctbal3$	$[\$c\_acctbal3 < p_1]$
$\$c\_acctbal4$	$[\$c\_acctbal4 < p_1]$
$\$l\_price1$	$[\$aggsum\_1 = 5 \times \$l\_price1]$
$\$l\_price5$	$[\$aggsum\_2 = 3 \times \$l\_price5]$

Table 5 Output of HAVING clause (1 tuple)

o_date	SUM(l_price)
$\$o\_date1$	$\$aggsum1$

output group. Then the second pre-group (with $\$o\_date2$) which includes two tuples is assigned to the second output group. When the third pre-group (with $\$o\_date3$) which includes one tuple is being assigned to the first output group (because of round-robin), the aggregation operator replaces $\$o\_date3$ with $\$o\_date1$ in order to put the five tuples into the same group. Similarly, the aggregation operator replaces $\$o\_date4$ from the input tuple with $\$o\_date2$. For the aggregation function, each output group g_i invokes the Simple Aggregation Operator mentioned early in this section with all the symbols that participated in the aggregation function as input, and gets a new symbol agg_{g_i} as output. Finally, for each group, the operator constructs a new symbolic tuple $\langle g_i, agg_{g_i} \rangle$ and returns it to the parent. Table 4(i) shows the output of the aggregation operator, and Table 4(ii) shows the updated PTable after the aggregation in the running example. Furthermore, since the aggregation operator involves attributes o_date and l_price, the Orders table and the Lineitem table are also updated (Fig. 5c shows the updated tables).

HAVING and Single GROUP-BY Attribute In most cases, dealing with a HAVING clause is the same as dealing with a selection.
Figure 5c shows the PTable content after the HAVING clause. It imposes two more constraints: $[\$aggsum1 \geq p2]$ which is the positive tuple and $[\$aggsum2 < p2]$ which is the negative tuple, and it returns Table 5 to the parent.

There is a special case for the aggregation operator together with the HAVING clause. When there is more than one parameter in the query which influences the number of tuples of each output group implicitly, it is necessary to ask the user to define the count of each output group explicitly. However, this special case rarely happens in practice and no queries in the TPC-H benchmark pose this behavior. Nonetheless, for completeness, QAGen also deals with this special case by proposing two different algorithms for pre-grouped and not pre-grouped attributes. However, due to space limit, we omit the detail here and refer to the reader to [28]

Multiple GROUP-BY Attributes If there is a set of GROUP-BY attributes G, the implementation of the aggregation operator depends not only on whether the input is pre-grouped, but also depends on whether the GROUP-BY attributes in the input have a tree-structure or have a graph-structure (see Sect. 3). QAGen currently supports queries with tree-structure GROUP-BY attributes (see Fig. 4).

The aggregation operator treats aggregation with multiple GROUP-BY attributes in the same way as the case of a single GROUP-BY attribute (Sect. 4.2.4). Assume attribute a_n is the attribute in G which is functionally dependent on the least number of other attributes in G. The aggregation operator treats a_n as the single GROUP-BY attribute and sets the rest of the attributes in A to a constant value v (attribute a_n is selected because it has the largest number of distinct symbols in the input comparing to the other attributes).

As an example, assume the following table is an input to an aggregation operator.

b	c	d
$\$b_1$	$\$c_1$	$\$d_1$
$\$b_2$	$\$c_1$	$\$d_1$
$\$b_3$	$\$c_2$	$\$d_1$

Assume the set of GROUP-BY attributes A is $\{b, c, d\}$, and the functional dependencies which hold on the input of the aggregation operator are: $\{b\} \rightarrow \{c, d\}$ and $\{c\} \rightarrow \{d\}$. According to the definition in Sect. 3, the set of GROUP-BY attributes G has a tree-structure.

In the input above, attribute b is functionally dependent on no attributes where d is functional dependent on b and c. As a result, the aggregation operator treats attribute b as the single GROUP-BY attribute and invoke the single GROUP-BY aggregation implementation. Other attributes use the same symbol for all input tuples (e.g., set all symbols for attribute c to be $c1$).

Since the aggregation operator with multiple-group attributes is handled by the aggregation operator that supports a single GROUP-BY attribute, it shares the same special cases (HAVING clause on top on an aggregation where the parameter values control the group count) as the case of aggregation with a single GROUP-BY attributes.

4.2.5 Symbolic execution of the projection operator

Symbolic execution on a projection operator is exactly the same as the traditional query processing, it projects the spec-

Table 6 Output of π(1 tuple)

SUM(l_price)
$*aggsum*1

ified attributes and no additional constraints are added. As a result, the final projection operator in the running example takes in the input from Table 5 and ends with the result shown in Table 6. However, the current version of QAGen has not supported the DISTINCT keyword yet and we will handle this keyword in our next version.

4.2.6 Symbolic execution of the union operator

In SQL, the UNION operator eliminates duplicate tuples if they exist. On the other hand, the UNION ALL operator does not eliminate duplicates. In SQP, the query analyzer does not offer any knob to the user to tune the UNION ALL operation. Therefore, the symbolic execution of the UNION ALL operation is straightforward to implement: it reuses the UNION ALL operator in RDBMS and unions the two inputs into one.

For the UNION operation, in SQP, the query analyzer offers users the following knob:

Knob:	Output Cardinality c (optional; default value = size of R + size of S)

Let R and S be the inputs of the UNION operation which are not pre-grouped. The symbolic execution of the UNION operator controls the output as follows:

1. During its getNext() call, if the output cardinality has not yet reached c, then (a) get a tuple t from R (or from S alternatively); and (b) return t to its parent. However, during the getNext() call, if the output cardinality has reached c already, then process [Post-processing] below and return null to its parent.
2. [Post-processing] Fetch the remaining tuples R^- and S^- from inputs R and S, respectively, set the symbols in tuple R^- and S^- to have the same symbol as one of the returned tuples t in the previous step.

4.2.7 Symbolic execution of the MINUS operator

In SQL, the MINUS operator selects all distinct rows that are returned by the query on the left hand side but not by the query on the right hand side.

Let R and S be the non-pre-grouped inputs of the MINUS operation. In this case, the query analyzer offers users the following knob:

Knob:	Output Cardinality c (optional; default value = size of R)

The symbolic execution of the MINUS operator controls the output as follows:

1. During its getNext() call, if the output cardinality has not yet reached c, then (a) get a tuple r^+ from R, and; (b) return r^+ to its parent. However, during the getNext() call, if the output cardinality has reached c already, then process [Post-processing] below, and return null to its parent.
2. [Post-processing] Fetch a tuple r^- from R, fetch all tuples S^- from S, set the symbols in tuple $s^- \in S^-$ to have the same symbol as r^-.

4.2.8 Symbolic execution of the INTERSECT operator

Knob:	Output Cardinality c (optional; default value = size of R)

In SQL, the INTERSECT operator returns all distinct rows selected by both queries. Currently, QAGen supports INTERSECT with non-pre-grouped inputs. Let R and S be the input of the INTERSECT operator, the symbolic execution of the INTERSECT operator is as follows:

1. During its getNext() call, if the output cardinality has not yet reached c, then (a) get a tuple r^+ from R, and get a tuple s^+ from S; (b) set the symbols of s^+ as same as r^+ and return r^+ to its parent. However, during the getNext() call, if the output cardinality has reached c already, return null to its parent.

4.2.9 Symbolic execution of nested queries

Nested queries in SQP reuses the techniques in traditional query processing because queries can be unnested by using join operators [16]. In order to allow a user to have full control on the input, the user should rewrite nested queries into their unnested forms before inputting to the system. If the inner query and the outer query refer to the same table(s), then the query analyzer disables some knobs on operators that may allow a user to specify different constraints on the operators that work on the same table in both inner and outer query.

5 Data instantiator in QAGen

This section presents the details of the data instantiator in QAGen. The data instantiator is responsible for the final phase of the whole data generation process. It fetches the symbolic tuples from the symbolic database and uses a constraint solver (strictly speaking, the constraint solver is a deci-

sion procedure [12]) to instantiate concrete values for them. The constraint solver takes as input a propositional formula (remember that a predicate can be represented by a formula in propositional logic). It returns a set of concrete values for the symbols in the formula that satisfies all the input predicates and the actual data types of the symbols. If the input formula is unsatisfiable, the constraint solver returns an error. Such errors, however, cannot occur in this phase because we assume there are no contradicting knob values. A constraint solver call is an expensive operation. In the worst case, the cost of a constraint solver call is exponential to the size of the input formula [8]. As a result, the objective of the data instantiator is to, if possible, minimize the number of calls to the constraint solver. Indeed, the predicate optimizations during SQP (e.g. reducing $\$aggsum = \$l\_price1 + \cdots + \$l\_price8$ to $\$aggsum = \$l\_price1 \times 8$) are designed for this purpose. After the data instantiator has collected all the concrete values of a symbolic tuple, it inserts the instantiated tuple into the final test database. The details of the data instantiator are as follows:

1. The process starts from any one of the symbolic tables.
2. It reads in a tuple t, say $\langle \$c\_id1, \$c\_acctbal1 \rangle$, from the symbolic tables.
3. [Look up symbol-to-value cache] For each symbol s in tuple t, (a) it first looks up s in a cache table called SymbolValueCache. SymbolValueCache stores the concrete values of the symbols that have been instantiated by the constraint solver; (b) if symbol s has been instantiated with a concrete value, then the symbol is initialized with the same cached value and then proceeds with the next symbol in t.
   In the running example, assume the constraint solver randomly instantiates the Customer table (4 tuples) first. Since symbol $\$c\_id1$ is the first symbol to be instantiated, it has no instantiated value stored in SymbolValueCache. However, assume later when instantiating the first two tuples of Orders table (with $\$o\_id1, \$o\_id2$), their o_cid values will use the same value as instantiated for $\$c\_id1$ by looking up SymbolValueCache.
4. [Instantiate values] Look up predicates P of s from the PTable. (a) If there are no predicates associated with s, then instantiate s by a unique value that is within the domain of s in input schema M.
   In the example, $\$c\_id1$ does not have any predicates associated with it (see PTable in Fig. 5). Therefore, the data instantiator does not instantiate s with a constraint solver but instantiates a unique value v (because c_id is a primary key), say, 1, to $\$c\_id1$. Afterwards, insert a tuple $\langle s, v \rangle$ (e.g., $\langle \$c\_id1, 1 \rangle$) into SymbolValueCache. (b) However, if s has some predicates P in the PTable, then compute the *predicate closure* of s. The predicate closure of s is computed by recursively looking up all the directly correlated or indirectly correlated predicates of s. For example, the predicate closure of $\$l\_price1$ is $[\$aggsum1 = 5 \times \$l\_price1$ AND $\$aggsum1 \geq p2]$. Then the predicate closure (which is in the form of a formula in propositional logic) is sent to the constraint solver (symbols that exist in SymbolValueCache are replaced by their instantiated values first). The constraint solver instantiates all symbols in the formula in a row (e.g., $\$l\_price1 = 10, \$aggsum1 = 50, p2 = 18$). For efficiency purposes, before a predicate closure is sent to the constraint solver, the data instantiator looks up another cache table called PredicateValuesCache. This cache table caches the instantiated values of predicates. Since many predicates in the PTable are similar in terms of their constraints they are capturing, the data instantiator only needs to store the query predicates stored in PredicateValuesCache. For example, predicates $[\$c\_acctbal1 \geq p1]$ and $[\$c\_acctbal2 \geq p1]$ in Fig. 5c share the same query predicate: $[\$c\_acctbal \geq p1]$. As a result, after the instantiation of predicate $[\$c\_acctbal1 \geq p1]$, the data instantiator inserts an entry $\langle [c\_acctbal \geq p1], \$c\_acctbal1, p1 \rangle$ into PredicateValuesCache. When the next predicate closure $[\$c\_acctbal2 \geq p1]$ needs to be instantiated, the data instantiator looks up its query predicate in PredicateValuesCache; if its query predicate is found in PredicateValuesCache, then the data instantiator skips the instantiation of this predicate and reuses the instantiated value of $\$c\_acctbal1$ in SymbolValueCache for symbol $\$c\_acctbal2$ (same for $p1$).

The number of constraint solver calls is minimized by the introduction of the cache tables SymbolValueCache and PredicateValuesCache. Experiments show that this feature is crucial or otherwise generating a 1GB query-aware database takes weeks instead of hours. Finally, note that in Step 4a, if a symbol s has no predicate associated with it, the data instantiator assigns a value to s according to its domain and its related integrity constraints (e.g., primary keys). In general, those values can be assigned randomly or always use the same value. However, it is also possible to instantiate some extra data characteristics (e.g., distribution) for those symbols to test certain aspects of the query optimizer even though those the values of symbols would not affect the query results.

6 The framework

This section presents the DBMS feature testing framework. So far, the discussion of QAGen is restricted to having a complete test case as input and generating a query-aware test database as output. A test case, as shown in Fig. 1, has to

consist of an SQL query Q and a set of knob values defined on each query operator. In practice, the most tricky job is to determine different sets of knob values for the test query in order to form different useful test cases. Currently, the knob values of a test case are manually chosen by the testers. The framework includes a tool to automate this step.

In software engineering, the test case selection problem is a way of forming a test suite for a program by generating test cases with different combinations of parameter values [1]. One way of choosing the values of a parameter is called the Category Partition (CP) method [31]. The CP method suggests the tester first partitions the domain of a parameter into subsets (called *partitions*) based on the assumption that all points in the same subset result in a similar behavior from the test object. The tester should select one value from each partition to form the set of parameter values.

Consider a simple query $R \bowtie S$ joining two tables R and S. Assume table R has 1,000 tuples and table S has 2,000 tuples and the two tables are not connected by foreign-key constraints. In this case, the values for the output cardinality knob for the join could be formed by partitioning the possible knobs values into, say, four partitions: Extreme case partition (0 tuple), Minimum case partition (1 tuple), Normal case partition (500 tuples), and Maximum case partition (1,000 tuples). In addition, Uniform distribution and Zipf distribution can be regarded as two partitions of the join key distribution knob.

Having decided the set of values for each parameter (knob), the next step is to combine those values to form different test cases (i.e., a test suite). There are various algorithms (known as combination strategies) for combining the parameter values and forming different test suites. Each algorithm will produce a test suite that achieves a certain *coverage*. One well-known coverage is called *Each-used* coverage (a.k.a. *1-wise* coverage). It requires every parameter value of every parameter to be included in at least one test case in the test suite. Consider a program with three parameters A, B and C and their respective sets of parameter values $\{a1, a2\}$,$\{b1, b2\}$ and $\{c1, c2\}$. An example test suite that satisfies the *Each-used* coverage is shown in Fig. 10a, which includes two test cases T_1 and T_2. Another classical coverage is *Pair-wise* coverage (a.k.a. *2-wise* coverage). It requires that every possible pair of intersecting values of any two parameters is included in some test cases in the test suite. Consider

$$T_1 : \{A = a1, B = b1, C = c1\}$$
$$T_2 : \{A = a2, B = b2, C = c2\}$$

(a) Each-used Coverage

$$T_1 : \{A = a1, B = b1, C = c1\}$$
$$T_2 : \{A = a1, B = b2, C = c2\}$$
$$T_3 : \{A = a1, B = b1, C = c2\}$$
$$T_4 : \{A = a2, B = b1, C = c1\}$$
$$T_5 : \{A = a2, B = b2, C = c2\}$$
$$T_6 : \{A = a2, B = b2, C = c1\}$$

(b) Pair-wise Coverage

Fig. 10 Coverage example

$$T_1 : \{\sigma_1 = 1(min), \bowtie = 1(min), \sigma_2 = 1(min)\}$$
$$T_2 : \{\sigma_1 = 1(min), \bowtie = 1(max), \sigma_2 = 1(min)\}$$
$$T_3 : \{\sigma_1 = 1(min), \bowtie = 2000(max), \sigma_2 = 2000(max)\}$$
$$T_4 : \{\sigma_1 = 1000(max), \bowtie = 1(min), \sigma_2 = 1(min)\}$$
$$T_5 : \{\sigma_1 = 1000(max), \bowtie = 2000(min), \sigma_2 = 2000(max)\}$$
$$T_6 : \{\sigma_1 = 1000(max), \bowtie = 1(max), \sigma_2 = 1(min)\}$$

Fig. 11 A pair-wise test suite generated by current combination strategies

the same example program as above, an example test suite that satisfies the *Pair-wise* coverage is shown in Fig. 10b, which includes six test cases. Other classic coverage includes *T-wise* [36], *Variable strength* [11], and *N-wise* coverage and each coverage criterion has its own pros and cons and they are served for different types of applications. There are different combination strategies to generate test suites that satisfy different coverage criteria. For example, the AETG algorithm [10] is a non-deterministic algorithm that generates test suites which satisfy the *Pair-wise* coverage.[5] As another example, the *Each Choice* algorithm [1] is a deterministic algorithm that generates test suites which satisfy the *Each-used* coverage. However, there are two problems that make it impossible to directly apply these algorithms in our automatic testing framework.

The first problem is that the knobs are correlated to each other in a knob-annotated QAGen execution plan. As a result, it is not easy to do category partitioning. As an example, it is difficult to partition the cardinality of the root (aggregation) operator of TPC-H Query 8 (see Fig. 14a) because the interesting value of the maximum case partition (i.e., the maximum number of output groups) depends on the cardinalities of its child operators.

The second problem is that the correlation of operators in a knob-annotated QAGen execution plan causes existing combination strategies to generate test suites that may not satisfy the coverage criterion. For example, consider a select-join query $\sigma_1(R) \bowtie \sigma_2(S)$ where R has 1,000 tuples and S has 2,000 tuples, and S has a foreign key referring to R on the join attribute. Assume that we are able to determine the minimum and the maximum cardinality of each operator:

	min	max
σ_1	1	1,000
σ_2	1	2,000
$\bowtie$	1	2,000

Then, according to the existing *Pair-wise* test suite combinational strategies, a test suite like the one in Fig. 11 will be returned. However, if we look closer into the test suite, we find that the generated test suite actually does not strictly fulfill the *Pair-wise* criterion. For test case T_1 and T_2, the selec-

[5] Non-deterministic algorithms means that it may generate different test suites every time.

tions on R and S return 1 tuple. Consequently, no matter the output cardinality of the join is defined as the minimum case partition (T_1) or the maximum case partition (T_2), the join can only return 1 tuple. As a result, T_1 and T_2 are the same and the final test suite does not make sure every possible pair of interesting values of any two knobs is included.

To automate the task of creating a set of meaningful test cases, it is necessary to devise a new set of combination strategies for each coverage that avoid the above problems. In the following, a simple method for generating *1-wise* test suites is presented. Discussion on how to design different combination strategies that satisfy different coverages would be an interesting research topic for the software engineering community.

One reason for using *1-wise* coverage in the framework is that there may be many knobs available in a QAGen query execution plan. Defining coverage stronger than *1-wise* (e.g., *2-wise*) may then result in a very large test suite. In addition, based on *1-wise* coverage, it is possible to design an algorithm so that the knob values are not affected by the correlations of the output cardinalities between operators in a query.

The following shows the test case generation algorithm. It takes as input a knob-annotated query plan and returns a set of test cases.

1. [Creating a test case for each cardinality *1-wise* partition] For each partition g of the output cardinality knob, create a temporary test case T_g.
2. [Assigning *1-wise* value to distribution knob] For each temporary test case T_g, create a test case T_{gd} from T_g using a distribution knob value d. The value d should not be repeated until each value is used once at least.
3. [Assigning real values to the cardinality partition] For each test case T_{gd}, parse test query Q of T_{gd} in a bottom-up manner and assign cardinality values to T_{gd} according to Table 7.

Figure 12 shows the test case generation process of a simple query $\sigma(R) \bowtie S$. In the current framework, we only consider the minimum and the maximum partitions for the cardinality knob and only Zipf and Uniform distribution for the distribution knob. Although the test generation algorithm is simple, experimental results show that the generated test suite can effectively generate different query-aware test databases that show different system behaviors of a commercial database system. In this work, we regard this simple SQL test case generation algorithm as a starting point for this new SQL test case generation problem. There are two points worth to notice here. First, the test case generation algorithm does not allow the same table to be used twice in the input of a binary operator, for example, the query $R \bowtie R$ is prohibited. Second, Table 7 does not capture the cases of pre-grouping input and the cases of having two disjoint subqueries [15] for a binary operator.

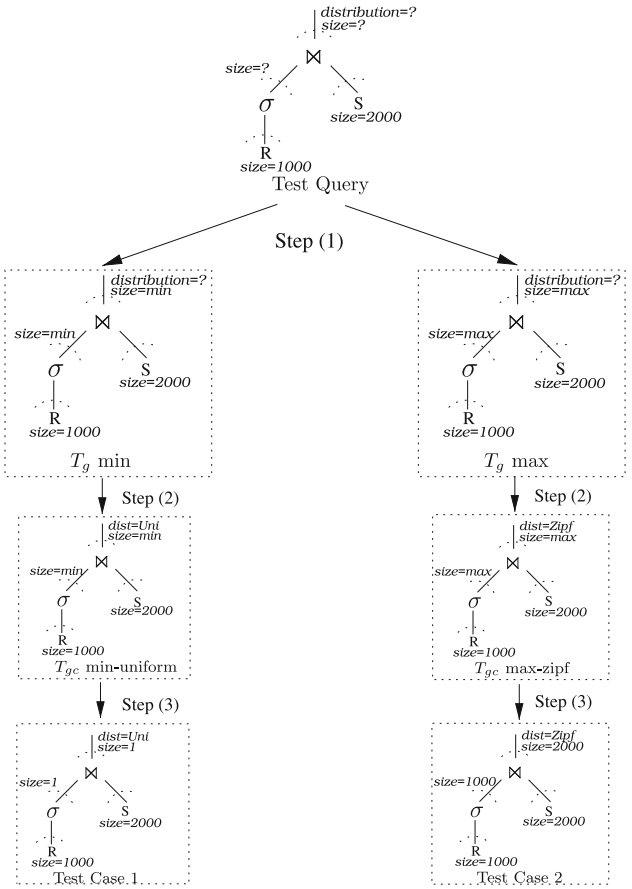

Fig. 12 Test case generation example

Table 7 Knob value table for the minimum and maximum partitions (The notation used in the table follows the discussion in Sect. 4. For example, R denotes the input of an unary operator and $|R|$ denotes its cardinality)

Operator	Minimum partition	Maximum partition
Selection	1	$\|R\|$
Aggregation	1	$\|R\|$
Join	1	$\|S\|$
Union	$\max(\|R\|, \|S\|)$	$\|R\| + \|S\|$
Minus	$\|R\| - \|S\|$	$\|R\|$
Intersect	1	$\min(\|R\|, \|S\|)$

Figure 13 shows the automatic DBMS feature testing framework. It is an extension of the QAGen architecture in Fig. 3. As usual, the tester gives a parameterized query Q and the schema M as input. After the query analyzing phase, the tester specifies the size of the base tables, and a test suite that satisfies the *1-wise* coverage is generated from the test suite generator. Each test case is then processed by the Symbolic

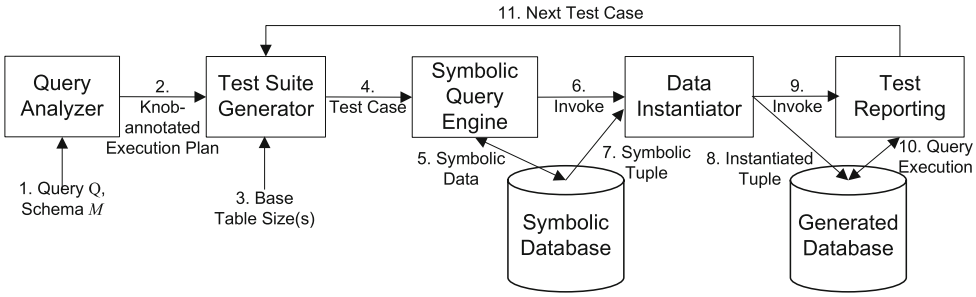

Fig. 13 The DBMS feature testing framework

Query Engine and the Data Instantiator and a query-aware test database is generated as usual. Finally, the test query of the test case is automatically executed against the generated database, and the execution details (e.g., the execution plan, cost, time) are inserted into the test report.

Note that, in general, testers use their domain knowledge in order to create input test queries. However, this step can also be automated by query generation tools (e.g., RAGS [34] and QGEN [32]). Furthermore, if the query analyzer detects that there are some operators with pre-grouped input or with disjoint subqueries in the query execution plan, it will prompt the tester to verify that automated generated test case before QAGen starts execution. As part of the future work, we plan to further improve the framework in order to eliminate these restrictions.

7 Experiments

We have run a set of experiments to evaluate our framework. The implementation is written in Java and it is installed on a Linux AMD Opteron 2.4 GHz Server with 6 GB of main memory. The symbolic database and the target database use PostgreSQL 8.1.11 and they are installed on the same machine. As a constraint solver, a publicly available constraint solver called Cogent [12] is used. Cogent is formally a decision procedure written in C. It takes as input a propositional formula and returns an instantiation of the variables in the formula if that is satisfiable. QAGen interacts with Cogent by writing the predicates to a text file and invokes Cogent through the Java `Runtime` class. QAGen then parses the output of Cogent (variable-value pairs) back into its internal representation. During the experiments, if the approximation ratio knob is enabled by the query analyzer, the value 0.1 is used.

We execute three sets of experiments with the following objectives: The first experiment (Sect. 7.1) studies the efficiency of the various operations in QAGen . The second experiment (Sect. 7.2) studies the performance of QAGen for generating databases in different sizes for different queries. The last experiment (Sect. 7.3) uses the testing framework to generate different test databases for the same query in order to study if the framework could effectively show different behavior of a commercial database system. In all experiments, all generated databases met the constraints (e.g., cardinality, approximation ratio) specified in the test cases.

7.1 Efficiency of QAGen operations

The objectives of this experiment are to evaluate the running time of individual QAGen operations and their scalability by generating three query-aware databases in different scales (10 M, 100 M, and 1 G). The input query is query 8 in the TPC-H benchmark. Its logical query plan is shown in Fig. 14a. We have chosen TPC-H query 8 because it is one of the most complex queries in TPC-H with 7-way joins and aggregations. This query has various input characteristics to the operators enabling us to evaluate the performance of different operator implementations (e.g., it involves both the normal equi-join and the special case of equi-join that needs solving the subset sum problem). The experiments are carried out as follows: first, three benchmark databases are generated using *dbgen* from the TPC-H benchmark. As a scaling factor, we use 10 MB, 100 MB, and 1 GB. Then, we execute query 8 on top of the three TPC-H databases, and collect the base table sizes and the cardinality of each intermediate result of each scale. The extracted cardinality of each intermediate result of query 8 is shown in Table 8 (Output-size) columns. Next, we generate three TPC-H-query-8-aware databases with the collected base table sizes and output cardinalities as input and measure the efficiency of QAGen for generating databases that produces the same cardinality results. The value distribution between two joining tables is uniform distribution.[6]

Table 8 shows the cost breakdown for generating query-aware databases for TPC-H query 8 in detail. QAGen takes less than 5 min to generate a 10 MB query-aware database. The SQP phase is fast and scales linearly. It takes about 2 min for a 10 MB database and about three hours for a 1G database. The longest SQP operation is the initialization of the large symbolic table Lineitem (#10 in Table 8), and the join between the intermediate result $R5$ and Lineitem (#11). This

[6] Note that the above procedure is for carrying out experiments only. Users of QAGen are expected to specify the cardinalities and approximation ratio by themselves.

Table 8 QAGen Execution time for TPC-H query 8

Symbolic query processing		size = 10 M		size = 100 M		size = 1 G	
#	Symbolic operation	Output-size	Time	Output-size	Time	Output-size	Time
1	Region	5	<1 s	5	<1 s	5	<1 s
2	σ(Region) = $R1$	1	<1 s	1	<1 s	1	<1 s
3	Nation	25	<1 s	25	<1 s	25	<1 s
4	($R1$ ⋈Nation) = $R2$	5	<1 s	5	<1 s	5	<1 s
5	Customer	1.5k	<1 s	15.0k	7 s	150k	61 s
6	($R2$⋈Customer) = $R3$	0.3k	1 s	3.0k	12 s	299.5k	85 s
7	Orders	15.0k	8 s	150.0k	87 s	1.5 m	923 s
8	σ(Orders) = $R4$	4.5k	12 s	45.0k	106 s	457.2k	1,040 s
9	($R3$ ⋈$R4$)=$R5$	0.9k	4 s	9.0k	37 s	91.2k	345 s
10	Lineitem	60.0k	34 s	600.5k	403 s	6001.2k	3,344 s
11	($R5$⋈Lineitem) = $R6$	3.6k	45 s	35.7k	441 s	365.1k	5,650 s
12	Part	2.0k	<1 s	20.0k	8 s	200k	89 s
13	σ(Part) = $R7$	12	1 s	147	13 s	1,451	111 s
14	($R7$ ⋈$R6$) = $R8$	29	5 s	282	45 s	2,603	762 s
15	Supplier	0.1k	<1 s	1k	<1 s	10k	3 s
16	(Supplier ⋈$R8$) = R9	29	<1 s	282	1 s	2,603	15 s
17	(Nation ⋈$R9$) = R10	29	<1 s	282	<1 s	2,603	5 s
18	χ($R8$) = $R11$	2	<1 s	2	1 s	2	9 s
Total SQP time		1 m : 53 s		19 m : 22 s		207 m : 27 s	
Data instantiation		size = 10 M		size = 100 M		size = 1 G	
19	Reading tuples from SDB	113 s		16 m : 04		169 m : 16 s	
20	Populating tuples into DB	29 s		4 m : 52 s		47 m : 51 s	
21	Cogent time/# calls/avg. # variables	3 s/14/1		3 s/14/1		53 s/14/1	
Total DI time		2 m : 26 s		23 m : 51 s		247 m : 33 s	
$\sum$		4 m : 19 s		43 m : 13 s		455 m : 0 s	

join takes a long time because it accesses the large Lineitem table frequently to update the symbolic values of the join attributes. In query 8, the input is pre-grouped on the last join (#17 in Table 8 and operator (17) in Fig. 14) and the approximation ratio knob is enabled. This join finishes quickly because the input size is not large. Based on this result, the performance of the proposed approximation scheme can be predicted by the detailed theoretical analysis given in [28]. Table 8 also shows that the symbolic execution of each individual operator scales well.

For TPC-H query 8, the data instantiation (DI) phase runs slightly longer than the SQP phase. It takes about 3 min to instantiate a 10 M query-8-aware database and about 4 hours to instantiate a 1G query-8-aware database. Nevertheless, indeed about 70% of the DI time is spent on reading the symbolic tuples from the symbolic tables and the PTable via JDBC (#19 in Table 8) for instantiation and about 20% of the DI time is spent on populating the instantiated tuples into the final database tables (#20). In the experiments, the number of constraint solver (cogent) calls is small – there are only 14 calls for the three scaling factors (#21 in Table 8). The number of calls is constant because the data instantiator extracts the query predicates for instantiation and therefore it does not increase with the number of tuples. Furthermore, #21 in Table 8 also reveals that the constraint formula sent by QAGen to Cogent has few variables (only one) and the time spent on Cogent is indeed significantly small. We repeat the same experiment by turning off the caching feature of QAGen, but it ends up that the data instantiation phase for a 1G database takes weeks instead of hours. This shows that the predicate optimization in SQP and the caching in the data instantiator work effectively.

7.2 Scalability of QAGen

The objective of this experiment is to evaluate the scalability of QAGen for generating a variety of query-aware test databases. Currently, QAGen supports 14 out of 22 TPC-H queries. It does not support some queries because some of them use non-equi-joins (e.g., Q17, Q22). Nevertheless, we

Table 9 QAGen scalability—TPC-H

	10 M	100 M	1 G
Q1			
Symbolic query processing	4 m : 24 s	37 m : 57 s	407 m : 32 s
Data instantiation	1 m : 59 s	18 m : 50 s	222 m : 07 s
$\sum$	6 m : 23 s	56 m : 47 s	629 m : 39 s
Q2			
DB overhead %	63%	62%	59%
Cogent time/calls/avg. variables	1 s/4/16	1 s/4/16	2 s/4/16
Symbolic query processing	14 s	2 m : 29 s	23 m : 24 s
Data instantiation	15 s	1 m : 44 s	19 m : 29 s
$\sum$	29 s	4 m : 13 s	42 m : 53 s
Q3			
DB overhead %	38%	54%	51%
Cogent time/calls/avg. variables	<1 s/8/29	1 s/8/29	1 s/8/29
Symbolic query processing	2 m : 16 s	23 m : 43 s	258 m : 16 s
Data instantiation	2 m : 29 s	25 m : 25 s	270 m : 32 s
$\sum$	4 m : 45 s	49 m : 08 s	528 m : 48 s
Q4			
DB overhead %	42%	44%	45%
Cogent time/calls/avg. variables	3 s/12/14.5	2 s/12/14.5	3 s/12/14.5
Symbolic query processing	3 m : 26 s	43 m : 14 s	404 m : 53 s
Data instantiation	2 m : 34 s	25 m : 29 s	283 m : 12 s
$\sum$	6 m : 0 s	68 m : 43 s	688 m : 05 s
Q6			
DB overhead %	61%	59%	60%
Cogent time/calls/avg. variables	2 s/8/14.6	2 s/8/14.6	2 s/8/14.6
Symbolic query processing	2 m : 32 s	25 m : 23 s	263 m : 50 s
Data instantiation	3 m : 0 s	31 m : 07 s	349 m : 43 s
$\sum$	5 m : 32 s	56 m : 30 s	613 m : 33 s
Q8			
DB overhead %	40%	40%	42%
Cogent time/calls/avg. variables	<1 s/4/16	1 s/4/16	1 s/4/16
Symbolic query processing	2 m : 01 s	19 m : 43 s	207 m : 27 s
Data instantiation	2 m : 33 s	24 m : 08 s	247 m : 33 s
$\sum$	4 m : 34 s	43 m : 51 s	455 m : 0 s
DB overhead %	40%	41%	42%
Cogent time/calls/avg. variables	2 s/14/1	3 s/14/1	3 s/14/1
Q9			
Symbolic query processing	3 m : 37 s	30 m : 06 s	320 m : 07 s
Data instantiation	2 m : 20 s	22 m : 42 s	249 m : 12 s
$\sum$	5 m : 57 s	52 m : 48 s	549 m : 19 s
DB overhead %	45.9%	51.45%	48%
Cogent time/calls/avg. variables	2 s/8/66	3 s/8/66	2 s/8/66
Q10			
Symbolic query processing	1 m : 45 s	19 m : 22 s	194 m : 10 s
Data instantiation	2 m : 36 s	25 m : 50 s	263 m : 22 s
$\sum$	4 m : 21 s	45 m : 12 s	457 m : 32 s
DB overhead %	39%	39%	38%
Cogent time/calls/avg. variables	2 s/8/14.8	3 s/8/14.8	3 s/8/14.8

Table 9 continued

	10 M	100 M	1 G
Q12			
Symbolic query processing	3 m : 30 s	35 m : 02 s	345 m : 17 s
Data instantiation	4 m : 19 s	45 m : 11 s	475 m : 38 s
$\sum$	7 m : 49 s	80 m : 13 s	820 m : 55 s
DB overhead %	39%	38%	40%
Cogent time/calls/avg. variables	1 s/4/18.2	1 s/4/18.2	2 s/4/18.2
Q14			
Symbolic query processing	1 m : 14 s	22 m : 29 s	225 m : 04 s
Data instantiation	2 m : 05 s	47 m : 11 s	464 m : 30 s
$\sum$	3 m : 19 s	69 m : 40 s	689 m : 34 s
DB overhead %	33%	30%	31%
Cogent time/calls/avg. variables	2 s/4/16.3	3 s/4/16.3	3 s/4/16.3
Q15			
Symbolic query processing	1 m : 30 s	23 m : 01 s	229 m : 38 s
Data instantiation	2 m : 07 s	46 m : 38 s	451 m : 56 s
$\sum$	3 m : 37 s	69 m : 39 s	681 m : 34 s
DB overhead %	36%	% 31	38%
Cogent time/calls/avg. variables	2 s/4/16	2 s/4/16	2 s/4/16
Q16			
Symbolic query processing	12 s	2 m : 17 s	45 m : 24 s
Data instantiation	14 s	1 m : 46 s	40 m : 53 s
$\sum$	28 s	4 m : 03 s	86 m : 17 s
DB overhead %	35%	40%	38%
Cogent time/calls/avg. variables	<1 s/8/28	2 s/8/28	3 s/8/28
Q18			
Symbolic query processing	2 m : 03 s	20 m : 22 s	204 m : 12 s
Data instantiation	2 m : 45 s	24 m : 18 s	271 m : 30 s
$\sum$	4 m : 48 s	48 m : 40 s	475 m : 49 s
DB overhead %	34%	32%	38%
Cogent time/calls/avg. variables	<1 s/4/18	1 s/4/18	2 s/4/18
Q19			
Symbolic query processing	2 m : 26 s	41 m : 06 s	429 m : 58 s
Data instantiation	2 m : 50 s	81 m : 13 s	739 m : 18 s
$\sum$	5 m : 16 s	122 m : 19 s	1,169 m : 16 s
DB overhead %	41%	32%	55%
Cogent time/calls/avg. variables	2 s/8/15.8	2 s/8/15.8	2 s/8/15.8

generate query-aware databases for the rest of the queries in three different scaling factors 10 M, 100 M and 1 G. Table 9 shows the detailed results. Experimental results show that both phases scale linearly for all 14 TPC-H queries. Consistent with the previous experiment, Table 9 shows that our techniques successfully minimize the overhead that spent on the expensive Cogent. First, the number of calls to Cogent is independent of the scaling factor. Second, the average number of variables in a constraint formula has been minimized to a level that only a minimum of 1 variable (Q8) and a maximum of 66 variables (Q9) exists in the constraint formula. As a result, Cogent does not exhibit its worst case behavior in all cases (at most 3 seconds are spent on Cogent). Table 9 shows that a large portion of the running time is spent on the interactions with the backend database: updating base symbolic tables (stored in PostgreSQL) during SQP, reading tuples from symbolic databases (essentially PostgreSQL) for data instantiation and inserting tuples into the final (PostgreSQL) database. Nevertheless, the overall running time is still fairly satisfactory: for most queries (Q1, Q3, Q4, Q6, Q8, Q9, Q10, Q12, Q14, Q15, Q18) a 1G useful query-aware test database can be obtained in about 10 hours (1 night); for some queries (Q2 and Q16) QAGen finishes in minutes and for an extreme case (i.e., Q19) QAGen finishes in 1 day. Com-

Table 10 Knob values and resulting execution plans

Result	TPC-H(Uniform/Zipf)	MIN-Uniform	MAX-Zipf
$R1$	1	1	5
$R2$	5	1	25
$R3$	3k	1	15k
$R4$	45k	1	150k
$R5$	9k	1	150k
$R6$	36k	1	600k
$R7$	147	1	20k
$R8$	282	1	600k
$R9$	282	1	600k
$R10$	282	1	600k
$R11$	2	1	2
Execution plan	Fig. 14b	Fig. 14c	Fig. 14d

pare with the databases generated by the existing database generators that offer little help to meaningful DBMS testing, testers just need to run QAGen on multiple machines for some nights, once per each test case. After that, they can re-use the useful query-aware test databases again and again in their subsequent testing phases.

7.3 Effectiveness of the DBMS feature testing framework

The objective of this experiment is to show how the test databases that are generated by the automatic testing framework can show different behavior of a DBMS. In this experiment, the target database size is fixed at 100 MB and the input query is query 8 in TPC-H. The experiments are carried out in the following way: First, we generate four query-aware databases for TPC-H query 8. Then, we execute query 8 on the four generated databases (on PostgreSQL) and study their physical execution plans. The first database [MIN-Uniform] is automatically generated by the testing framework using the minimum case partition. The database will let query 8 to have the minimum cardinality on each intermediate result during execution. In the [MIN-Uniform] database, the key values between two joining relations have a Uniform distribution. Furthermore, during a grouping operation, tuples will be uniformly distributed into different groups in the [MIN-Uniform] database. The second database [MAX-Zipf] is also generated by the test framework using the maximum case partition with a Zipf distribution. The third database [TPCH-Uniform] is manually added to the test suite and is generated by QAGen using the intermediate result sizes extracted from executing query 8 on TPC-H *dbgen* database (as in the first experiment above). The last database [TPCH-Zipf] is generated by QAGen using the same intermediate result sizes as [TPCH-Uniform] but with a Zipf distribution. Table 10 shows the intermediate result sizes of the above set up.

Figure 14 shows the physical execution plans of executing TPC-H query 8 on the generated query-aware databases. By controlling the output cardinalities of the operators, it causes PostgreSQL to use different join strategies. For example, when the cardinality of each output is minimum [MIN-Uniform], PostgreSQL tends to use a left-deep-join order (Fig. 14c). When the cardinality of each output is maximal [MAX-Zipf], PostgreSQL tends to use a bushy-tree join order (Fig. 14d). The output cardinalities also strongly influence the choice of physical operators; when the output cardinality is large, PostgreSQL tends to use hash joins (Fig. 14d). However, when the output cardinality is small, PostgreSQL tends to use fewer hash joins but used sort-merge-joins and nested-loop-joins (Fig. 14b,c). The input and output cardinality also influence the choice of physical aggregation operators. When the input to the aggregation (i.e., $R10$ in Table 2) is minimum or same as the TPC-H size, then PostgreSQL tends to use group aggregation (Fig. 14b,c). However, when the input to is maximum, then PostgreSQL tends to first do a hash aggregation and then sort it (Fig. 14d).

Controlling the distributions of the query operators shows that the operators in PostgreSQL are less sensitive to the data distribution. For example, when the cardinality is same as TPC-H size (Fig. 14b), the distribution knob does not influence the execution plans. Moreover, the distribution knob also has less influence on the choice of physical operators.

In this experiment, we attempt to use other database generation tools to generate the same set of test databases which can produce the same intermediate query results. We try to run this experiment with two commercial test database generators, DTM Data Generator and IBM DB2 Test Database Generator, and one research prototype [21].[7] However, these tools only allow constraining the base tables properties and

[7] We also attempt to evaluate the tools [4,5] from Microsoft, however their tools are not publicly available.

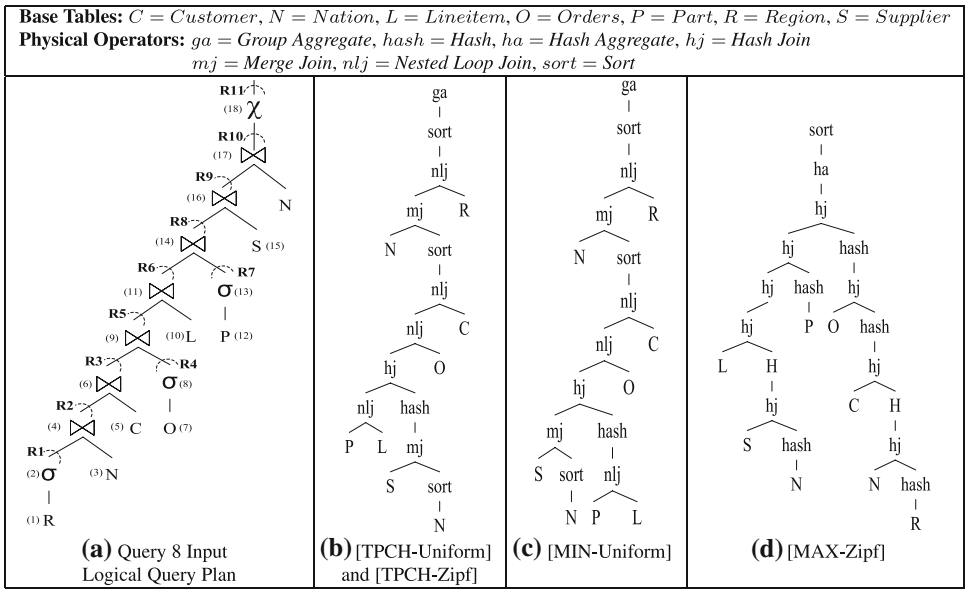

Fig. 14 **a** Input Query **b–d** execution plans of TPC-H query 8

we fail to manually control the intermediate result sizes for the purposes of this experiment.

8 Related work

The closest related work in DBMS testing is the work of [5,30] which studies the generation of query parameters for test queries with given test databases. However, existing database generation tools such as IBM DB2 Database Generator and others (e.g., [4,19,21]) were designed to generate general-purpose test databases and do not take account of test queries, and thus the generated databases cannot guarantee sufficient coverage of specific test cases. As a consequence, [5,30] may hardly find a good database to work on and only SPJ queries are supported.

QAGen extends symbolic execution [25] and proposes the concept of SQP to generate query-aware databases. SQP is related to constraint databases (e.g., [27]); however, constraint databases focus on constraints that represent infinite concrete data (e.g., spatial-temporal data) whereas SQP works on finite but abstract data. Recently, [2] also studied the problem of query-aware test database generation. In particular, based on the work in [2,29] proposed the concept of reverse query processing, which takes as input an application query and the corresponding query result, and returns a corresponding database instance. The focus of reverse query processing is to generate minimal size test databases for functional tests of *database applications*, which cannot control the intermediate query results for the purpose of testing DBMS features.

The automatic testing framework in this paper is related to a number of software testing research work. For example, [1] first states the test case selection problem for traditional program testing. Some solutions for the traditional test case selection problem can be found in [1,10,11,20,36].

9 Conclusions and future work

This work presented a framework for DBMS feature testing. The framework includes a test database generator QAGen that generates tailor-made test databases for different DBMS test cases. QAGen is based on SQP, a technique that combines traditional query processing and symbolic execution from software engineering. It has shown that QAGen is able to generate query-aware databases for complex queries and it scales linearly. By using the framework, test cases for testing the features of a DBMS can be constructed automatically and test reports can be obtained with minimal human effort.

QAGen is a test automation tool that runs in the background. Although the test databases are not generated in seconds, each test case only generates a test database *once* and then the generated test database can be used for testing many times. For benchmarking, however, it would be advantageous to generate a *single* test database (or at least a minimal number of test databases if a minimal or single test database is not doable) that can cover multiple test queries. We are in the process of extending QAGen for performance testing. The basic idea of this new version of QAGen is as follows: Given a number of test cases $T_1, T_2, \ldots, T_n$, we first symbolically process each test case T_i separately (without data instantiation) and obtain n individual symbolic database instances $SDB_1, SDB_2, \ldots, SDB_n$. Then, we design a symbolic database integrator to "merge" the symbolic database instances in best-effort. After the "merging" step, we re-use QAGen's data instantiator to instantiate the symbolic databases with real values. The time consuming data instantiation step is

invoked only once per merged benchmark database. Therefore, the test generation process is fast and a handful of benchmark databases can be obtained.

As requested by some industry partners, we are also in the process of extending QAGen to generate query-aware databases with duplicate data values and null values, which both have impacts on various database components. Another interesting future direction is to extend the current testing framework so that it supports more coverage criteria. For example, it would be interesting if the framework could generate test cases where an operator (e.g. selection) gets a maximum partition input but returns a minimum partition output. Finally, we believe the work on SQP can be integrated with traditional symbolic execution so as to extend program verification and test case generation techniques to support database applications.

Acknowledgments We thank the anonymous VLDBJ reviewers for their insightful comments. We also thank Heikki Mannila and Marc Nunkesser for providing useful feedback on various part on this work.

References

1. Ammann, P., Offutt, J.: Using formal methods to derive test frames in category-partition testing. In: Annual Conference on Computer Assurance, pp. 69–80 (1994)
2. Binnig, C., Kossmann, D., Lo, E.: Reverse query processing. In: ICDE (2007)
3. Binnig, C., Kossmann, D., Lo, E., Özsu, M.T.: QAGen: generating query-aware test databases. In: SIGMOD, pp. 341–352 (2007)
4. Bruno, N., Chaudhuri, S.: Flexible database generators. In VLDB, pp. 1097–1107 (2005)
5. Bruno, N., Chaudhuri, S., Thomas, D.: Generating Queries with Cardinality Constraints for DBMS Testing. TKDE (2006)
6. Chaudhuri, S., Narasayya, V.: TPC-D data generation with skew. Accessible at http://ftp.research.microsoft.com/users/viveknar/tpcdskew (1997)
7. Chordia, S., Dettinger, E., Triou, E.: Different query verification approaches used to test entity sql. In: DBTest, p. 7 (2008)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking (2000)
9. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
10. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The aetg system: an approach to testing based on combinatiorial design. IEEE TSE **23**(7), 437–444 (1997)
11. Cohen, M.B., Gibbons, P.B., Mugridge, W.B., Colbourn, C.J.: Constructing test suites for interaction testing. In: ICSE, pp. 38–48 (2003)
12. Cook, B., Kroening, D., Sharygina, N.: Cogent: Accurate theorem proving for program verification. In: CAV, pp. 296–300 (2005)
13. DTM Data Generator. http://www.sqledit.com/dg/
14. Elhemali, M., Giakoumakis, L.: Unit-testing query transformation rules. In: DBTest, p. 3 (2008)
15. Elkan, C.: A decision procedure for conjunctive query disjointness. In: PODS (1989)
16. Ganski, R.A., Wong, H.K.T.: Optimization of nested SQL queries revisited. In: SIGMOD, pp. 23–33 (1987)
17. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness (1990)
18. Graefe, G.: Query evaluation techniques for large databases. ACM Comput. Surv. **25**(2), 73–170 (1993)
19. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly generating billion-record synthetic databases. In: SIGMOD, pp. 243–252 (1994)
20. Grindal, M., Offutt, J., Andler, S.F.: Combination testing strategies: a survey. Softw. Test. Verif. Reliab. **15**(3), 167–199 (2005)
21. Houkjær, K., Torp, K., Wind, R.: Simple and realistic data generation. In: VLDB, pp. 1243–1246 (2006)
22. Ibarra, O.H., Kim, C.E.: Fast approximation algorithms for the knapsack and sum of subset problems. J. ACM **22**(4), 463–468 (1975)
23. IBM DB2 Test Database Generator. http://www-306.ibm.com/software/data/db2imstools/db2tools/db2tdbg/
24. Kellerer, H., Mansini, R., Pferschy, U., Speranza, M.G.: An efficient fully polynomial approximation scheme for the subset-sum problem. J. Comput. Syst. Sci. **66**(2), 349–370 (2003)
25. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
26. Klug, A.: Calculating constraints on relational expression. TODS **5**(3), 260–290 (1980)
27. Kuijpers, B.: Introduction to constraint databases. SIGMOD Rec. **31**(3), 35–36 (2002)
28. Lo, E., Binnig, C., Kossmann, D., Ozsu, M.T., Hon, W.-K.: A Framework for Testing DBMS Features. Technical Report. Hong Kong Polytechnic University (2008)
29. Mannila, H., Räihä, K.-J.: Test data for relational queries. In: PODS, pp. 217–223 (1986)
30. Mishra, C., Koudas, N., Zuzarte, C.: Generating targeted queries for database testing. In: SIGMOD, pp. 499–510 (2008)
31. Ostrand, T.J., Balcer, M.J.: The category-partition method for specifying and generating fuctional tests. Commun. ACM, **31**(6) (1988)
32. Poess, M., Stephens, J.M.: Generating thousand benchmark queries in seconds. In: VLDB, pp. 1045–1053 (2004)
33. Przydatek, B.: A fast approximation algorithm for the subset-sum problem. Int. Trans. Oper. Res. **9**(4), 437–459 (2002)
34. Slutz, D.R.: Massive Stochastic Testing of SQL. In: VLDB, pp. 618–622 (1998)
35. Stephens, J.M., Poess, M.: Mudd: a multi-dimensional data generator. In: WOSP, pp. 104–109 (2004)
36. Williams, A.W., Probert, R.L.: A measure for component interaction test coverage. In: AICCSA, pp. 304–312 (2001)
37. Zipf, G.: Human Behaviour and the Principle of Least Effort (1949)