Autom Softw Eng
DOI 10.1007/s10515-014-0158-y

Program-input generation for testing database applications using existing database states

Kai Pan · Xintao Wu · Tao Xie

Received: 25 August 2013 / Accepted: 16 June 2014

Abstract Testing is essential for quality assurance of database applications. Achieving high code coverage of the database applications is important in testing. In practice, there may exist a copy of live databases that can be used for database application testing. Using an existing database state is desirable since it tends to be representative of real-world objects' characteristics, helping detect faults that could cause failures in real-world settings. However, to cover a specific program-code portion (e.g., block), appropriate program inputs also need to be generated for the given existing database state. To address this issue, in this paper, we propose a novel approach that generates program inputs for achieving high code coverage of a database application, given an existing database state. Our approach uses symbolic execution to track how program inputs are transformed before appearing in the executed SQL queries and how the constraints on query results affect the application's execution. One significant challenge in our problem context is the gap between program-input constraints derived from the program and from the given existing database state; satisfying both types of constraints is needed to cover a specific program-code portion. Our approach includes novel query

K. Pan
Outlook Service of Applications and Services Group, Microsoft Corporation,
Redmond, WA 98052, USA
e-mail: kaipan@microsoft.com

X. Wu
Department of Computer Science and Computer Engineering, University of Arkansas at Fayetteville,
Fayetteville, AR 72701, USA
e-mail: xintaowu@gmail.com

T. Xie (✉)
Department of Computer Science, University of Illinois at Urbana-Champaign,
Urbana, IL 61801, USA
e-mail: taoxie@illinois.edu

Published online: 08 July 2014

formulation to bridge this gap. We incorporate the data-instantiation component in our framework to deal with the case that no effective program input values can be attained. We determine how to generate new records and populate them in the new database state such that the code along the path can be covered. We also extend our approach of program-input generation to test database applications including canonical queries and group-by queries. Our approach is loosely integrated into Pex, a state-of-the-art white-box testing tool for .NET from Microsoft Research. Empirical evaluations on two real database applications show that our approach assists Pex to generate program inputs that achieve higher code coverage than the program inputs generated by Pex without our approach's assistance.

Keywords Database application testing · Test generation · Dynamic symbolic execution

1 Introduction

Database applications are ubiquitous, and it is critical to assure high quality of database applications. To assure high quality of database applications, testing is commonly used in practice. Testing database applications can be classified as functional testing, performance testing (load and stress, scalability), security testing, environment and compatibility testing, and usability testing. Among them, functional testing aims to verify the functionality of the code under test. An important task of functional testing is to generate test inputs to achieve full or at least high code coverage, such as block or branch coverage of the database application under test. For database applications, test inputs include both program inputs (i.e., input parameters) and database states. Typically, a database application communicates with the associated database through four steps. First, the application sets up a connection with the database. Second, it constructs a query to be executed and combines the query into the connection. Third, if the query's execution yields an output, the result is returned. Fourth, the returned query result is manipulated for further exploration. In database applications, program input values are often combined into the executed query (either directly or after a chain of computations) and record values in the returned query result set are often directly or indirectly involved in path conditions. Hence producing appropriate test inputs to achieve high code coverage faces great challenges as database states play crucial roles in database application testing and constraints from embedded SQL queries and queries' result sets are correlated with program code.

1.1 Illustrative example

The example code snippet shown in Fig. 1 includes a portion of C# source code from a database application that calculates some statistics related to mortgages. The corresponding database contains two tables: `customer` and `mortgage`. Their schema-level descriptions and constraints are given in Table 1. The `calcStat` method described in the example code snippet receives two program inputs: `type` that determines the years of mortgages and `zip` that indicates the zip codes of customers. A variable `fzip`

Fig. 1 An example code snippet from a database application under test

```
01:public int calcStat(int type,int zip) {
02:  int years = 0, count = 0, totalBalance = 0;
03:  int fzip = zip + 1;
04:  if (type == 0)
05:     years = 15;
06:  else
07:     years = 30;
08:  SqlConnection sc = new SqlConnection();
09:  sc.ConnectionString = "..";
10:  sc.Open();
11:  string query = "SELECT C.SSN, C.income,"
    +" M.balance FROM customer C, mortgage M"
    +" WHERE M.year='" + years +"' AND"
    +" C.zipcode='"+ fzip + "' AND C.SSN = M.SSN";
12:  SqlCommand cmd = new SqlCommand(query, sc);
13:  SqlDataReader results = cmd.ExecuteReader();
14:  while (results.Read()){
15:     int income = int.Parse(results["income"]);
16:     int balance = int.Parse(results["balance"]);
17:     int diff = income - 1.5 * balance;
18:     if (diff > 100000){
19:        count++;
20:        totalBalance = totalBalance + balance;}}
21:  return totalBalance;}
```

Table 1 Database schema

Customer table			Mortgage table		
Attribute	Type	Constraint	Attribute	Type	Constraint
SSN	Int	Primary key	SSN	Int	Primary key
Zipcode	String	[1, 99999]			Foreign key
Name	Int		Year	Int	
Gender	String				
Age	Int	(0, 100)	Balance	Int	(1000, max)
Income	Int				

is calculated from `zip` and in our example `fzip` is given as "`zip`+1". We use `fzip = zip+1` to represent the scenario that program input values are combined into the executed query after a chain of computations. Then the database connection is set up (Lines 08–10). The database query is constructed (Line 11) and executed (Lines 12 and 13). The tuples from the returned result set are iterated (Lines 14–20). For each tuple, a variable `diff` is calculated from the values of the `income` field and the `balance` field. If `diff` is greater than `100000`, a counter variable `count` is increased (Line 19) and `totalBalance` is updated (Line 20). The method finally returns the calculation result.

Both program inputs and database states are crucial in testing this database application because (1) the program inputs determine the embedded SQL statement in Line 11; (2) the database states determine whether the true branch in Line 14 and/or the true branch in Line 18 can be covered, being crucial to functional testing, because covering a branch is necessary to expose a potential fault within that branch; (3) the database

states also determine how many times the loop body in Lines 14–20 is executed, being crucial to performance testing.

1.2 Problem formalization

In practice, there may exist a copy of live databases that can be used for database application testing. Using an existing database state is desirable since it tends to be representative of real-world objects' characteristics, helping detect faults that could cause failures in real-world settings. However, it often happens that a given database with an existing database state (even with millions of records) returns no records (or returns records that do not satisfy branch conditions in the subsequently executed program code) when the database receives and executes a query with arbitrarily chosen program input values. For example, method `calcStat` takes both `type` and `zip` as inputs. To cover a path where conditions at Lines 14 and 18 are both `true`, we need to assign appropriate values to variables `years` and `fzip` so that the execution of the SQL statement in Line 12 with the query string in Line 11 will return non-empty records, while at the same time attributes `income` and `balance` of the returned records also satisfy the condition in Line 18. Since the domain for program input `zip` is large, it is very likely that, if a tester enters an arbitrary `zip` value, execution of the query on the existing database will return no records, or those returned records do not satisfy the condition in Line 18. Hence, it is crucial to generate program input values such that test inputs with these values can help cover various code portions when executed on the existing database.

1.3 Proposed solution

To address this issue, in this paper, we propose a novel approach that generates program inputs for achieving high code coverage of a database application, given an existing database state. In our approach, we first examine close relationships among program inputs, program variables, branch conditions, embedded SQL queries, and database states. For example, program variables used in the executed queries may be derived from program inputs via complex chains of computations (we use `fzip=zip+1` in our illustrative example) and path conditions involve comparisons with record values in the query's result set (we use `if (diff>100000)`) in our illustrative example). We then automatically generate appropriate program inputs via executing a formulated auxiliary query on the given database state.

In particular, our approach uses dynamic symbolic execution (DSE) (Sen et al. 2005) to track how program inputs to the database application under test are transformed before appearing in the executed queries and how the constraints on query results affect the later program execution. We use DSE to collect various intermediate information.

Our approach addresses one significant challenge in our problem context: there exists a gap between program-input constraints derived from the program and those derived from the given existing database state; satisfying both types of constraints is needed to cover a specific program-code portion. During DSE, these two types of constraints cannot be naturally collected, integrated, or solved for test generation. To

address this challenge, our approach, which was first presented in Pan et al. (2011b), includes novel query formulation to bridge this gap. In particular, based on the intermediate information collected during DSE, our approach automatically constructs new auxiliary queries from the SQL queries embedded in code under test. The constructed auxiliary queries use those database attributes related with program inputs as the target selection and incorporate those path constraints related with query result sets into the selection condition. After the new auxiliary queries are executed against the given database, we attain effective program input values for achieving code coverage.

It may happen that the corresponding queries cannot retrieve specific results (e.g., required to achieve code coverage) from the current database no matter how we choose program input values. The reason is that the existing database state may not contain necessary records that satisfy those collected constraints to cover various paths. In that case, we extend our approach to generate new records and populate them in the new database state such that the code along a path can be covered.

SQL queries embedded in application program code could be very complex. For example, they may involve nested subqueries with aggregation functions, union, distinct, and group-by views, etc. A large body of work exists on query transformation in databases (e.g., Seshadri et al. 1996; Dayal 1987; Kim 1982), where various decorrelation techniques were used to unnest complex queries into equivalent single level canonical queries. In this paper, we extend our approach on auxiliary query construction to attain effective program input values when canonical queries are embedded. Queries with group-by and aggregation are widely used for online analytical processing in decision support systems. In this paper, we also extend out approach to deal with queries with group-by and aggregation.

This paper makes the following main contributions:

1. The first problem formalization for program-input generation given an existing database state to achieve high code coverage.
2. An approach of program-input generation based on symbolic execution and query formulation for bridging the gap between program-input constraints from the program and from the given existing database state. We incorporate the data-instantiation component in our framework to deal with the case that no effective program input values can be attained. We also extend our approach of program-input generation to handle canonical queries and group-by queries.
3. Evaluations on two real database applications to assess the effectiveness of our approach upon Pex (Microsoft Research Foundation of Software Engineering Group 2007), a state-of-the-art white-box testing tool for .NET from Microsoft Research. Empirical results show that our approach assists Pex to generate program inputs that achieve higher code coverage than the program inputs generated by Pex without our approach's assistance.

The rest of this paper is organized as follows. We revisit dynamic symbolic execution for database application testing in Sect. 2. In Sect. 3, we present our approach that can assist DSE to determine appropriate program input values such that high code coverage can be achieved using the existing database state. We also present our approach to deal with aggregate calculation in program code and show data instantiation when no effective program input values can be attained. In Sect. 4, we extend our approach of

Table 2 A given database state

Customer table						Mortgage table		
SSN	Zipcode	Name	Gender	Age	Income	SSN	Year	Balance
001	27695	Alice	Female	35	50000	001	15	20000
002	28223	Bob	Male	40	150000	002	15	30000

program-input generation to handle canonical queries and group-by queries. We show our empirical evaluations on two real database applications in Sect. 5 and present related work in Sect. 6. Finally we offer our concluding remarks and discuss future work in Sect. 7.

2 Dynamic symbolic execution in database application testing

Recently, DSE (Sen et al. 2005) was proposed for test generation. Various tools for different languages have been developed such as Pex (Microsoft Research Foundation of Software Engineering Group 2007) for .NET programs. (e.g., C Sen et al. 2005, Java Emmi et al. 2007, and C# Microsoft Research Foundation of Software Engineering Group 2007). In traditional symbolic execution (King 1976), a program is executed symbolically with symbolic inputs rather than concrete inputs. DSE extends traditional symbolic execution by running a program with concrete inputs while collecting both concrete and symbolic information at runtime, making the analysis more precise. DSE first starts with default or arbitrary inputs and executes the program concretely. Along the execution, DSE simultaneously performs symbolic execution to collect symbolic constraints on the inputs obtained from predicates in conditions. DSE flips a branch condition and conjuncts the negated branch condition with constraints from the prefix of the path before the branch condition. DSE then feeds the conjuncted conditions to a constraint solver to generate new inputs to explore not-yet-covered paths. The whole process terminates when all the feasible program paths have been explored or the number of explored paths has reached the predefined upper bound.

DSE has also been used in testing database applications (Emmi et al. 2007; Taneja et al. 2010). Emmi et al. (2007) developed an approach for automatic test generation based on DSE. Their approach uses a constraint solver to solve collected symbolic constraints to generate both program input values and corresponding database records. The approach involves running the program simultaneously on concrete program inputs as well as on symbolic inputs and a symbolic database. In the first run, the approach uses random concrete program input values, collects path constraints over the symbolic program inputs along the execution path, and generates database records such that the program execution with the concrete SQL queries can cover the current path. To explore a new path, it flips a branch condition and generates new program input values and corresponding database records.

We use our preceding example shown in Fig. 1 to illustrate how DSE works and show its limitations of generating appropriate test inputs to achieve high code coverage. We assume that we have an existing database state shown in Table 2. We use Pex

(Microsoft Research Foundation of Software Engineering Group 2007), a DSE tool for .NET, in our following discussions. Pex is a DSE tool that can enable parameterized unit testing, an extension of unit testing that reduces test maintenance costs. Pex provides a set of APIs that help access intermediate information of its DSE process. In our evaluation, to get the intermediate information collected by Pex, we mainly call its API methods under class PexSymbolicValue such as GetPathConditionString() and GetRelevantInputNames(). Pex integrates the constraint solver Z3.[1] Z3 is a high-performance theorem prover being developed at Microsoft Research. The constraint solver Z3 supports linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, uninterpreted functions, and quantifiers. We also use ZQL[2] as our SQL parser. ZQL can parse all the ANSI queries (including SELECT, INSERT, UPDATE, DELETE, COMMIT, ROLLBACK, SET TRANSACTION) and is not tailored for any specific database management systems. ZQL parses SQL and fills in Java structures representing SQL statements and expressions. Following its provided API methods, we can get the intermediate and final information about the parsed SQL statements.

To run the program for the first time against the existing database state, Pex uses default values for program inputs `type` and `zip`. In this example, because `type` and `zip` are both integers. Pex simply chooses "`type`=0, `zip`=0" as default values. The condition in Line 04 is then satisfied and the query statement with the content in Line 11 is dynamically constructed. In Line 12 where the query is executed, we dynamically get the concrete query string as

```
Q1: SELECT C.SSN, C.income, M.balance
    FROM customer C, mortgage M
    WHERE M.year=15 AND C.zipcode=1 AND C.SSN=M.SSN
```

Through static analysis, we also get `Q1`'s corresponding abstract form as

```
Q1abs: SELECT C.SSN, C.income, M.balance
       FROM customer C, mortgage M
       WHERE M.year=: years AND C.zipcode=: fzip
             AND C.SSN=M.SSN
```

The execution of `Q1` on Table 2 yields zero record. Thus, the `while` loop body in Lines 14–20 is not entered and the exploration of the current path is finished. We use the Pex API method `PexSymbolicValue.GetPathConditionString()` after Line 14 to get the path condition along this path:

```
P1:(type == 0) && (results.Read() != true)
```

To explore a new path, Pex flips a part of the current path condition from "`type == 0`" to "`type != 0`" and generates new program inputs as "`type=1, zip=0`". The condition in Line 04 is then not satisfied and the SQL statement in Line 11 is dynamically determined as

```
Q2: SELECT C.SSN, C.income, M.balance
    FROM customer C, mortgage M
    WHERE M.year=30 AND C.zipcode=1 AND C.SSN=M.SSN
```

[1] http://research.microsoft.com/en-us/um/redmond/projects/z3/.

[2] http://zql.sourceforge.net/.

Note that here we have the same abstract form for Q2 as for Q1. However, the execution of `Q2` still returns zero record, and hence the execution cannot enter the `while` loop body either. The path condition for this path is

```
P2:(type == 1) && (results.Read() != true)
```

We see that at this point no matter how Pex flips the current collected path condition, it fails to explore any new paths. Since Pex has no knowledge about the `zipcode` distribution in the database state, using the arbitrarily chosen program input values often incurs zero returned records when the query is executed against the existing database state. As a result, none of the paths involving the `while` loop body could be explored. In testing database applications, previous test-generation approaches (e.g., Emmi et al. 2007) then invoke constraint solvers to generate new records and instantiate a new test database state.

However, by looking into the existing database state as shown in Table 2, we see that if we use an input like "`type=0, zip=27694`", the execution of the query in Line 11 will yield one record {`C.SSN = 001,C.income = 50000,M.balance = 20000`}, which further makes Line 14 condition `true` and Line 18 condition `false`. Therefore, using the existing database state, we are still able to explore this new path:

```
P3:(type == 0) && (results.Read() == true) &&(diff <= 100000)
```

Furthermore, if we use "`type=0, zip=28222`", the execution of the query in Line 11 will yield another record {`C.SSN = 002,C.income = 150000,M.balance = 30000`}, which will make both Line 14 condition and Line 18 condition `true`. Therefore, we can explore this new path:

```
P4:(type == 0) && (results.Read() == true) &&(diff > 100000)
```

In Sect. 3, we present our approach that can assist Pex to determine appropriate program input values such that high code coverage can be achieved using the existing database state.

3 Approach

Our approach differs from Emmi et al.'s (2007) approach in that we leverage DSE as a supporting technique to generate effective program input values by exploiting the existing database state. As a result, high code coverage of the application can be achieved without generating new database states. Our approach assists DSE to determine appropriate program input values such that the executed query can return sufficient records to cover various code portions.

Suppose that a program takes a set of parameters $I = \{I_1, I_2, \ldots, I_k\}$ as program inputs and contains one embedded SQL query. Throughout this section, we assume the SQL query takes the simple form:

```
SELECT  C1, C2, ..., Ch
FROM    from-list
WHERE   A1 AND A2 ... AND An
```

In the SELECT clause, there is a list of h strings where each may correspond to a column name or with arithmetic or string expressions over column names and constants following the SQL syntax. In the FROM clause, there is a `from-list` that consists of a list of tables. We assume that the WHERE clause contains n predicates, $A = \{A_1, A_2, \ldots, A_n\}$, connected by $n-1$ "AND"s. Each predicate A_i is of the form *expression op expression*, where *op* is a comparison operator (=, <>, >, >=, <, <=) or a membership operator (IN, NOT IN) and *expression* is a column name, a constant or an (arithmetic or string) expression. Throughout this paper, we use subscript i to denote an arbitrary variable. Note that here we assume that the WHERE clause contains only conjunctions using the logical connective "AND". We discuss how to process complex SQL queries in Sect. 4.

3.1 Auxiliary query construction and program input generation

The major idea of our approach is to construct an auxiliary query based on the intermediate information (i.e., the executed query's concrete string and its abstract form, symbolic expressions of program variables, and path conditions) collected by DSE. Our strategy is to rewrite the executed query by selecting program inputs' associated database attributes from the database. The constraints of the program inputs' associated database attributes in our auxiliary queries come from two parts: the executed query constructed dynamically by path conditions before the query is executed, and the returned result set's associated path conditions after the query is executed.

The program contains program variables and some program variables may be data-dependent on program inputs I. We say a program variable is data-dependent on a program input if this variable directly or indirectly refers to the program input's value. When a program variable is data-dependent on a program input that is turned symbolic during symbolic execution, the variable will be represented with a symbolic expression with the occurrence of the symbolic variable corresponding to the program input. We denote program variables data-dependent on program inputs I as $V = \{V_1, V_2, \ldots, V_t\}$. We denote program variables that directly retrieve the values from the query's returned result set as $U = \{U_1, U_2, \ldots, U_w\}$. Program variables in U may be involved in path conditions after query's execution. As illustrated in our example, not-covered branches or paths are usually caused by the empty returned result set (e.g., for path P1) or insufficient returned records that cannot satisfy later executed conditions (e.g., for path P3). Specifically we have three cases:

- Case 1: some predicate expressions in the WHERE clause of Q may involve comparisons with program variables in $V = \{V_1, V_2, \ldots, V_t\}$ that are data-dependent on program inputs I.
- Case 2: program variables in branch conditions after executing the query may be data-dependent on the query's result set.
- Case 3: program variables in branch conditions after executing the query may be data-dependent on program inputs I.

Algorithm 1 shows our pseudo procedure to construct the auxiliary query $\tilde{Q}$. The algorithm accepts as inputs a simple SQL query in its both concrete form Q and abstract form Q_{abs}, program inputs $I = \{I_1, I_2, \ldots, I_k\}$, and the current path condition $PC = pc_1 \wedge pc_2 \wedge \cdots \wedge pc_s$. During its path exploration, DSE flips a branch

Algorithm 1 Auxiliary Query Construction

Input: a simple query Q, Q's abstract form Q_{abs},
program input set $I = \{I_1, I_2, ..., I_k\}$,
path condition $PC = pc_1 \wedge pc_2 \wedge ... \wedge pc_s$
Output: an auxiliary query $\tilde{Q}$

Find variables $V = \{V_1, V_2, ..., V_t\}$ data-dependent on I;
Decompose Q_{abs} with a SQL parser to derive SELECT, FROM, and WHERE clauses;
Construct a predicate set $A = \{A_1, A_2, ..., A_n\}$ from Q's WHERE clause;
Construct an empty predicate set $\tilde{A}$, an empty attribute set C_V, and an empty simple query $\tilde{Q}$;
for each predicate $A_i \in A$ **do**
  if A_i does not contain program variables **then**
    Leave A_i unmodified and check the next predicate;
  else
    if A_i does not contain program variables from V **then**
      Substitute A_i's program variables with their corresponding concrete values in Q;
    else
      Substitute the variables from V with the expression expressed by I;
      Substitute the variables not from V with their corresponding concrete values in Q;
      Copy A_i to $\tilde{A}$;
      Add A_i's associated database attributes to C_V;
    end if
  end if
end for
Append C_V to $\tilde{Q}$'s SELECT clause;
Copy Q's FROM clause to $\tilde{Q}$'s FROM clause;
Append $A - \tilde{A}$ to $\tilde{Q}$'s WHERE clause;
Find variables $U = \{U_1, U_2, ..., U_u\}$ coming directly from Q's result set;
Find U's corresponding database attributes $C_U = \{C_{U1}, C_{U2}, ..., C_{Uw}\}$;
for each branch condition $pc_i \in PC$ after Q's execution **do**
  if pc_i contains variables data-dependent on U **then**
    if pc_i does not contain variables data-dependent on V **then**
      Substitute the variables in pc_i with the expression expressed by the variables from U;
      Substitute the variables from U in pc_i with U's corresponding database attributes in C_U;
      Add the branch condition in pc_i to $\tilde{PC}$;
    else
      Leave pc_i unchanged and check the next branch condition;
    end if
  end if
end for
Append all the branch conditions in $\tilde{PC}$ to $\tilde{Q}$'s WHERE clause;
return $\tilde{Q}$;

condition pc_i (e.g., one executed after the query execution) from the false branch to the true branch to cover a target path. Such flipping derives a new constraint or path condition for the target path. DSE feeds this constraint to the constraint solver to generate a new test input. Our algorithm is triggered when the later execution with the new test input does not cover the true branch of pc_i as planned, likely due to database interactions along the path.

3.1.1 Query dependent on program inputs

In the path exploration, DSE keeps records of all program variables and their concrete and symbolic expressions in the program along this path. From the records, we deter-

mine program variables $V = \{V_1, V_2, \ldots, V_t\}$ that are data-dependent on program inputs I (Line 1). We decompose Q_{abs} using a SQL parser called ZQL and get its n predicates $A = \{A_1, A_2, \ldots, A_n\}$ from the WHERE clause (Lines 2 and 3). We construct an empty predicate set $\tilde{A}$ for the auxiliary query $\tilde{Q}$ (Line 4). Lines 5–21 present how to construct the clauses (SELECT, FROM, and WHERE) of the auxiliary query $\tilde{Q}$.

In Lines 5–18, we examine the relationship between each predicate $A_i \in A$ and program variables in V. If A_i does not contain any program variable, we leave A_i unchanged and check the next predicate (Lines 6 and 7). If A_i contains program variable from V, we then check whether any contained program variable comes from the set V. If no program variables in the predicate are from V, we substitute them with their corresponding concrete values in Q (Lines 9 and 10). If some program variables contained in the predicate come from V (indicating the predicate is data-dependent on program inputs), we substitute them with their symbolic expressions (expressed by the program inputs in I) (Line 12), substitute all the other program variables that are not from V with their corresponding concrete values in Q (Line 13), copy the predicate A_i to $\tilde{A}$ (Line 14), and add A_i's associated database attributes to a temporary attribute set C_V (Line 15). Those attributes in C_V will be included in the SELECT clause of the auxiliary query $\tilde{Q}$. After processing all the predicates in A, we get an attribute set $C_V = \{C_{V1}, C_{V2}, \ldots, C_{Vj}\}$ and a predicate set $\tilde{A} = \{\tilde{A_1}, \tilde{A_2}, \ldots, \tilde{A_l}\}$. Note that here all the predicates in $\tilde{A}$ are still connected by the logical connective "AND". The attributes from C_V form the attribute list of the $\tilde{Q}$'s SELECT clause (Line 19). All the predicates in A - $\tilde{A}$ connected by "AND" form the predicates in the $\tilde{Q}$'s WHERE clause (Line 21). Note that the `from-list` of the $\tilde{Q}$'s FROM clause is the same as that of Q (Line 20).

We take the path P3 (Line 04 `true`, Line 14 `true`, and Line 18 `false`) in our preceding example shown in Fig. 1 to illustrate our algorithm. The program input set is I = {`type, zip`} and the path condition PC is

```
P3:(type == 0) && (results.Read() == true)
   &&(diff <= 100000)
```

The program variable set V is {`type,zip,fzip`}. When flipping the condition `diff<=100000`, Pex fails to generate satisfiable test inputs for the flipped condition `diff > 100000`. The abstract form is shown as

```
Qabs:  SELECT C.SSN, C.income, M.balance
       FROM  customer C, mortgage M
       WHERE M.year=: years AND
             C.zipcode=: fzip AND C.SSN=M.SSN
```

We see that the predicate set A in the WHERE clause is formed as {`M.year=:years, C.zipcode=:fzip, C.SSN=M.SSN`}. Predicates `M.year=:years` and `C.zipcode=:fzip` contain program variables `years` and `fzip`, respectively. Because the predicate `M.year=:years` does not contain any program variable from V, we substitute `years` with its concrete value in Q and the predicate expression is changed to `M.year=15`. In contrast, the predicate `C.zipcode=:fzip` contains the program variable `fzip`, which belongs to V. We replace `fzip` with `zip+1` and the new predicate becomes

`C.zipcode=:zip+1`. The associated attribute `C.zipcode` is added to $\tilde{A}$ and is also added in the SELECT clause of the auxiliary query $\tilde{Q}$. After processing all the predicates in A, $\tilde{A}$ is `C.zipcode=:zip+1`, $A - \tilde{A}$ is `M.year=15 AND C.SSN=M.SSN`, and the attribute set C_V is `C.zipcode`. The constructed auxiliary query $\tilde{Q}$ is:

```
SELECT  C.zipcode
FROM    customer C, mortgage M
WHERE   M.year=15 AND C.SSN=M.SSN
```

When executing the preceding auxiliary query against the existing database state, we get two `zipcode` values, `27695` and `28223`. The corresponding program input `zip` can take either `27694` or `28222` because of the constraint `{C.zipcode=: zip+1}` in our example. A test input with the program input either "`type=0, zip=27694`" or "`type=0, zip=28222`" can guarantee that the program execution enters the `while` loop body in Lines 14–20. However, there is no guarantee that the returned record values satisfy later executed branch conditions. For example, if we choose "`type=0, zip=27694`" as the program input, the execution can enter the `while` loop body but still fails to satisfy the branch condition (i.e., `diff>100000`) in Line 18. Hence it is imperative to incorporate constraints from later branch conditions into the constructed auxiliary query.

3.1.2 Branch conditions after query execution dependent on the result set

Program variables in branch condition $pc_i \in PC$ after executing the query may be data-dependent on returned record values. In our example, the value of program variable `diff` in branch condition "`diff > 100000`" is derived from the values of the two variables `income`, `balance` that correspond to the values of attributes `C.income`, `M.balance` of returned records. Lines 22–34 in Algorithm 1 show how to incorporate later branch conditions in constructing the WHERE clause of the auxiliary query.

We first get the set of program variables $U = \{U_1, U_2, \ldots, U_w\}$ that directly retrieve the values from the query's returned result set, and treat them as symbolic inputs (Line 22). For each program variable U_i, we also keep its corresponding database attribute C_{Ui}. Note that here C_{Ui} must come from the columns in the query's SELECT clause. We save them in the set $C_U = \{C_{U1}, C_{U2}, \ldots, C_{Uw}\}$ (Line 23). In Lines 24–34, we examine whether branch conditions after Q's execution contain program variables that are either data-dependent on U (recall that U contains program variables that directly retrieve the values from the query's returned result set) or data-dependent on V (recall that V contains program variables that are data-dependent on program input I). Lines 27–29 show actions for the case where the branch condition pc_i contains variables that are only data-dependent on U (but not on V). We substitute such variables in pc_i with their symbolic expressions with respect to the symbolic input variables from U (Line 27) and replace each U_i in pc_i with its corresponding database attribute C_{Ui} (Line 28). The modified pc_i is then added in $\tilde{PC}$ (Line 29). After examining all branch conditions, we append $\tilde{PC}$ to the $\tilde{Q}$'s WHERE clause (Line 35).

In our example, the variable set U is `income, balance` and its corresponding attribute set C_U is `C.income, M.balance`. The host variable `diff` in the conditional "`diff > 100000`" is derived from the variables in U. We substitute `diff` with

```
01:public int calcStat(int type,int zip, int inputDiff) {
     ...
14:  while (results.Read()){
     ...
18:    if (diff > inputDiff){
19:       count++;
20:       totalBalance = totalBalance + balance;}}
21:  return totalBalance;}
```

Fig. 2 Example code where a program input also appears in branch conditions after query execution

`income - 1.5 * balance` and then replace the variables `income` and `balance` with the database attributes `C.income` and `M.balance`. The modified branch condition `C.income-1.5*M.balance>100000` is finally appended to the WHERE clause, and the new auxiliary query is

```
SELECT  C.zipcode
FROM    customer C, mortgage M
WHERE   M.year=15 AND C.SSN=M.SSN AND
        C.income - 1.5 * M.balance > 100000
```

When executing the preceding auxiliary query against the existing database state, we get the `zipcode` value as "28223". Having the constraint `C.zipcode=:zip+1`, input "`type=0, zip=28222`" can guarantee that the program execution enters the true branch in Line 18.

3.1.3 Branch conditions after query dependent on program inputs

Branch conditions after query execution could also be data-dependent on program inputs besides the returned record values. Consider the modified example code in Fig. 2 where we have another input `inputDiff` (Line 1) and the original condition `diff > 100000` is replaced with `diff > inputDiff` (Line 18). We observe that to make the condition in Line 18 `true`, we need to set appropriate values for `inputDiff` so that the value of the variable `diff` derived from query's returned records is greater than `inputDiff`.

Lines 30 and 31 of Algorithm 1 deal with the case that the branch condition pc_i within the query-result-manipulation code are data-dependent on both program inputs and the query's result set. Recall that the constructed auxiliary query from Lines 1-21 is

```
SELECT  C.zipcode
FROM    customer C, mortgage M
WHERE   M.year=15 AND C.SSN=M.SSN
```

Since the variables (e.g., `diff` in Line 18) involved in such branch conditions have been assigned with concrete values during DSE, DSE could automatically set appropriate values to those related program inputs (e.g., `inputDiff` in Line 18). In our example, using the derived input `zip = 28222` as illustrated in Sect. 3.1, program execution could enter the `while` loop in Line 14 in Fig. 2. At this point, during DSE, the variable `diff` in Line 18 is assigned with concrete value `20000` (diff = 50000 - 1.5 * 20000).

Algorithm 2 Program Input Generation

Input: an auxiliary query $\tilde{Q}$, program inputs I
intermediate results C_V and $\tilde{A}$ from Algorithm 1
Output: program input values R for I

1: Execute $\tilde{Q}$ against the given database and retrieve returned values R_V for the attributes in C_V;
2: Substitute the attributes C_V for predicates in $\tilde{A}$ with the values in R_V, resulting in new predicates in $\tilde{A}$;
3: Feed the new predicates in $\tilde{A}$ to a constraint solver and derive final values R for I;
4: **return** Output final program input values R;

Then, DSE could set a value for `inputDiff` directly to satisfy the branch condition in Line 18 (e.g., `inputDiff = 19999`).

3.1.4 Program input generation

Algorithm 2 shows our pseudo procedure of program-input generation given the constructed auxiliary query $\tilde{Q}$ and intermediate results C_V and $\tilde{A}$ from Algorithm 1. Note that the predicates $\tilde{A}$ keep the symbolic expressions of C_V with respect to program inputs I. In Line 1, we execute the auxiliary query $\tilde{Q}$ against the existing database state and return a set of values R_V for attributes in C_V. Each attribute in C_V can be traced back to some program variable in $V = \{V_1, V_2, \ldots, V_t\}$. Recall that V contains program variables that are data-dependent on program inputs I. We substitute the attributes C_V with their corresponding concrete values in R_V resulted from executing $\tilde{Q}$ against the existing database state and have new predicates in $\tilde{A}$ for program inputs I (Line 2). We then feed these new predicates in $\tilde{A}$ to a constraint solver to derive the values for program inputs I (Line 3).

In our illustrative example, after executing our auxiliary query on Table 2, we get a returned value "28223" for the attribute `C.zipcode`. In $\tilde{A}$, we have `C.zipcode=:zip+1`. After substituting `C.zipcode` in `C.zipcode=:zip+1` with the value "28223", we have `28223=:zip+1`. The value "28222" for the program input `zip` can then be derived by invoking a constraint solver.

In practice, the result R could be a set of values. For example, the execution of the auxiliary query returns a set of satisfying zip code values. If multiple program input values are needed, we can repeat the same constraint solving process to produce each returned value in R.

3.2 Dealing with aggregate calculation

Up to now, we have investigated how to generate program inputs through auxiliary query construction. Our algorithm exploits the relationships among program inputs, program variables, executed queries, and path conditions in source code. Database applications often deal with more than one returned record. In many database applications, multiple records are iterated from the query's returned result set. Program variables that store values retrieved from the returned result set further take part in aggregate calculations. The aggregate values then are used in the path condition. In this section, we discuss how to capture the desirable aggregate constraints on the result

set returned for one or more specific queries issued from a database application. These constraints play a key role in testing database applications but previous work (Chays et al. 2008; Deng and Chays 2005) on generating database states has often not taken them into account.

Consider the following code after the query's returned result set has been iterated in our preceding example shown in Fig. 1:

```
...
14:  while (results.Read()){
15:     int income = int.Parse(results[``income'']);
16:     int balance = int.Parse(results[``balance'']);
17:     int diff = income - 1.5 * balance;
18:     if (diff > 100000){
19:        count++;
20:        totalBalance = totalBalance + balance;}}
20a: if (totalBalance > 500000)
20b:  do other calculation...
21:  return ...;}
```

Here, the program variable `totalBalance` is data-dependent on the variable `balance` and thus is associated with the database attribute `M.balance`. The variable `totalBalance` is involved in a branch condition `totalBalance > 500000` in Line 20a. Note that the variable `totalBalance` is aggregated from all returned record values. For simple aggregate calculations (e.g., sum, count, average, minimum, and maximum), we are able to incorporate the constraints from the branch condition in our auxiliary query formulation. Our idea is to extend the auxiliary query with the GROUP BY and HAVING clauses. For example, DSE automatically identifies that `totalBalance` and `M.balance` are data-dependent. We then manually derive that the variable `totalBalance` is a summation of all the values from the attribute `M.balance`. The variable `totalBalance` can be transformed into an aggregation function `sum(M.balance)`. We include `C.zipcode` in the GROUP By clause and `sum(M.balance)` in the HAVING clause of the extended auxiliary query:

```
SELECT C.zipcode,sum(M.balance)
FROM  customer C, mortgage M
WHERE M.year=15 AND C.SSN=M.SSN
      AND C.income - 1.5 * M.balance > 100000
GROUP BY C.zipcode
HAVING sum(M.balance) > 500000
```

3.2.1 Cardinality constraints

In many database applications, we often require the number of returned records to meet some conditions. For example, after execution reaches Line 20, we may have another piece of code appended to Line 20 as

```
20c:  if (count >= 3)
20d:    computeSomething();
```

Here we use a special DSE technique (Godefroid and Luchaup 2011) for dealing with input-dependent loops. This technique applies simple loop-guard pattern-matching rules to derive an input constraint to capture the number of iterations of

input-dependent loops during dynamic symbolic execution. With this technique, we learn that the subpath with the conditions in Lines 14 and 18 being `true` has to be invoked at least three times in order to cover the branch condition `count >= 3` in Line 20c. Hence we need to have at least three records iterated into Line 18 so that true branches of Lines 14, 18, and 20c can be covered. In our auxiliary query, we simply add `COUNT(*) >= 3` in the HAVING clause to capture this cardinality constraint.

```
SELECT   C.zipcode
FROM     customer C, mortgage M
WHERE    M.year=15 AND C.SSN=M.SSN
         AND C.income - 1.5 * M.balance > 100000
GROUP BY C.zipcode
HAVING   COUNT(*) >= 3
```

Program logic could be far more complex than the appended code in Lines 20a-d of our example. We emphasize here that our approach up to now works for only aggregate calculations that are supported by the SQL built-in aggregate functions. When the logic iterating the result set becomes more complex than SQL's support, we cannot directly determine the appropriate values for program inputs. For example, some zipcode values returned by our auxiliary query could not be used to cover the true branch of Lines 20a, b because the returned records with the input zipcode values may fail to satisfy the complex aggregate condition in Line 20a. However, our approach can still provide a super set of valid program input values. Naively, we could iterate all the candidate program input values to see whether some of them can cover a specific branch or path. However, it may incur large cost when the returned set is large.

3.3 Modifying database state

Although constructing auxiliary queries provides a way of deriving effective program inputs based on the existing database state, executing the constructed auxiliary queries may still return an empty result set, which indicates that the current database state lacks qualified records. To deal with such situation, we need to generate new records and populate them back to the database.

For the preceding example code in Fig. 1, choosing Line 06 to be `true` will make true the condition `M.year = 30` for the WHERE clause from the query in Line 11, where we observe that the database in Table 2 does not contain sufficient records. Thus, to cover two new paths P5 and P6 as shown below, we need to have at least one record that satisfies constraints on the query result set.

```
P5:(type != 0) && (results.Read() == true) &&(diff <= 100000)
P6:(type != 0) && (results.Read() == true) &&(diff > 100000)
```

In our approach, we generate new records by invoking the data-instantiation component. We build symbolic databases consisting of symbolic tuples. We use *symbolic query processing* by parsing the SQL statement and substitute the symbolic tuples in the symbolic databases with symbols that reflect the constraints from the SQL statement.

Table 3a and c show the symbolic tables of `customer` and `mortgage`, respectively. For example, tuple $t1$ in Fig. 3(s) is a symbolic tuple of symbolic relation `customer`

Table 3 Symbolic query processing: $t1$ for path P5 and $t2$ for path P6

(a) Symbolic customer

	SSN	Zipcode	Income
$t1$	$a1	$b1	$c1
$t2$	$a2	$b2	$c2

(b) Symbolic customer after SQP

	SSN	Zipcode	Income
$t1$	$a1	$b1=:zip+1	$c1 − 1.5*$f1<=100000
$t2$	$a2	$b2=:zip+1	$c2 − 1.5*$f2>100000

(c) Symbolic mortgage

	SSN	Year	Balance
$t1$	$d1	$e1	$f1
$t2$	$d2	$e2	$f2

(d) Symbolic mortgage after SQP

	SSN	Year	Balance
$t1$	$d1=$a1	$e1=:years	$c1 − 1.5*$f1<=100000
$t2$	$d2=$a2	$e2=:years	$c2 − 1.5*$f2>100000

(e) Instantiated customer

	SSN	Zipcode	Income
$t1$	003	28223	50000
$t2$	004	28223	150000

(f) Instantiated mortgage

	SSN	Year	Balance
$t1$	003	30	10000
$t2$	004	30	20000

to cover path P5; symbol $a1 represents any value in the domain of attribute `SSN` and symbol $c1 represents any value in the domain of attribute `income`. Similarly, tuple $t2$ is a symbolic tuple to cover path P6. After symbolic query processing, the symbolic tables shown in Table 3b and d have captured all the constraint requirements specified in the symbolic case to cover paths P5 and P6 but without concrete data. We can see that $t1$ involves the constraint $\$c1 - 1.5 * \$f1 \leq 100000$ while $t2$ involves the constraint $\$c2 - 1.5 * \$f2 > 100000$. Note that dependencies on program inputs that are not database attributes will not be involved in data instantiation. Those constraints will be explored by DSE and the corresponding input values will be generated by DSE directly.

In our approach, we also manually collect basic constraints at the database schema level (e.g., not-NULL, uniqueness, referential integrity constraints, domain constraints, and semantic constraints) and incorporate them in data generation. In our prototype system, we assume that column values are always known because the null

values complicate many issues (including comparisons, logical connectives, and joins). Attribute `age` in table `customer` must be in the range (0, 100) and attribute `balance` must be greater than 1000. The symbolic tables together with the basic constraints are then sent to a constraint solver, which can instantiate the symbolic tuples with concrete values. Table 3e and f show the instantiated records from a constraint solver. For example, the record $t1$ is instantiated as "`C.SSN=003, C.zipcode=28223, C.income=50000`" in table `customer` and "`M.SSN=003, M.year=30, M.balance=10000`" in table `mortgage`. From constraints such as "`$b1=:zip+1`", the constraint solver also returns the value `28222` for input parameter `zip`. Hence, a test input with program input `{type=1, zip=28222}` on the newly populated database state can lead to coverage of path P5. Note that we restore the database state to its original state when we test a new path.

To cover the preceding paths P5 and P6 for the example code in Fig. 1, generating one new record for each path could be enough. However, in practice, satisfying cardinality constraints usually requires a large number of records. Cardinality constraints significantly affect the total cost of generating new database records. Consider the example related with cardinality constraints as discussed in Sect. 3.2. Suppose that we have another piece of code appended to Line 20 as

```
20c:  if (count >= 1000)
20d:    computeSomething();
```

Note that here we have a more extensive cardinality constraint `count >= 1000` for the qualified records than the previous cardinality constraint (i.e., `count >= 3`). As discussed in Sect. 3.2, we can first construct an auxiliary query as below to choose values for program input `zip`.

```
SELECT   C.zipcode
FROM     customer C, mortgage M
WHERE    M.year = 15 AND C.SSN = M.SSN
         AND C.income - 1.5 * M.balance > 100000
GROUP BY C.zipcode
HAVING   COUNT(*) >= 1000
```

However, executing this auxiliary query may return an empty result, indicating that the current existing database does not contain sufficient qualified records. Naively, we can generate at least 1000 new records from scratch and populate them back to the database, which incurs a high computational cost on the constraint solver. This is because generating a large number of records requires many times of expensive invocations of the constraint solver (one invocation per symbolic tuple). In practice, an empty result returned by the auxiliary query could be caused by the insufficient size of existing qualified records. For example, if the existing database has already contained 900 such records and in that case, we need to generate only 100 other new records, requiring much lower cost than generating all the 1000 records from scratch.

Based on the required cardinality constraints, to reduce the cost of generating new records from scratch, we conduct database record generation in the following way. First, we check whether we need to generate new records if the constructed auxiliary query returns an empty result. If no, our technique does not need to help with such situation; if yes, second, we check whether the constructed auxiliary query contains cardinality constraints. If so, we remove the required cardinality constraints and get

another auxiliary query by selecting the size of current qualified records. Third, we run this modified auxiliary query on the existing database and get the value for the size of current qualified records. Fourth, we compare the size with the required cardinality constraints. We can derive how many more records are needed and then conduct the data generation.

For example, when we observe that running the auxiliary query returns an empty result, we remove the cardinality constraints `HAVING COUNT(*) >= 1000`, add `COUNT(*)` to the SELECT clause, and get a new auxiliary query as

```
SELECT   C.zipcode, COUNT(*)
FROM     customer C, mortgage M
WHERE    M.year = 15 AND C.SSN = M.SSN
         AND C.income - 1.5 * M.balance > 100000
GROUP BY C.zipcode
```

Running this auxiliary query will return a value 900 for `COUNT(*)` if the existing database has already contained 900 qualified records within a specific zipcode. Comparing with the required cardinality constraints `COUNT(*) >= 1000`, we detect that at least 100 other records are needed. Hence, we generate new records by invoking the data-instantiation component. In this way, we can significantly reduce the cost when the current existing database has already contained a large number of qualified records and only a few more new records are needed.

4 Dealing with complex queries

SQL queries embedded in application program code could be very complex. The fundamental structure of a SQL query is a query block, which consists of SELECT, FROM, WHERE, GROUP BY, and HAVING clauses. If a predicate or some predicates in the WHERE or HAVING clause are of the form *expression op Q* where *Q* is a query block and *expression* is a column name, a constant or an (arithmetic or string) expression, the query is a *nested query*.

We present our algorithm of dealing with canonical queries in Sect. 4.1 and the algorithm of dealing with queries with group-by and aggregation in Sect. 4.2. We assume in our approach that queries do not contain nested subqueries. A large body of work exists on query transformation in databases (e.g., Seshadri et al. 1996; Dayal 1987; Kim 1982), where various decorrelation techniques were developed to unnest complex queries into equivalent single level canonical queries.

4.1 Canonical query

Generally, there are two types of canonical queries: DPNF with the WHERE clause consisting of a disjunction of conjunctions as shown below

```
SELECT  C1, C2, ..., Ch
FROM    from-list
WHERE   (A11 AND ... AND A1n) OR ...
        OR (Am1 AND ... AND Amn)
```

Algorithm 3 Program Input Generation for DPNF Query

Input: a DPNF query's abstract form Q_D, program inputs I
Output: program input value set R_D for I

1: Decompose Q_D with a SQL parser to get SELECT, FROM, and WHERE clauses;
2: **for** each disjunction D_i in Q_D's WHERE clause **do**
3: Build an empty query Q_i;
4: Append Q_D's SELECT clause to Q_i's SELECT clause;
5: Append Q_D's FROM clause to Q_i's FROM clause;
6: Append D_i to Q_i's WHERE clause;
7: Apply Algorithm 1 on Q_i and get its auxiliary query $\tilde{Q}_i$;
8: Apply Algorithm 2 on $\tilde{Q}_i$ and get output R_i;
9: $R_D = R_D \cup R_i$;
10: **end for**
11: **return** Output final program input values R_D;

and CPNF with the WHERE clause consisting of a conjunction of disjunctions (such as `(A11 OR ... OR A1n) AND ... AND (Am1 OR ... OR Amn)`). Note that DPNF and CPNF can be transformed mutually using DeMorgan's rules.

Algorithm 3 shows our pseudo procedure on how to formulate auxiliary queries and determine program input values given a general DPNF query. Our previous Algorithm 1 deals with only a special case of DPNF where the query's WHERE clause contains only one disjunction `A11 AND ... AND A1n`. The algorithm first decomposes Q_D with a SQL parser to get SELECT, FROM, and WHERE clauses (Line 1). The algorithm then builds m simple queries Q_i $(i = 1, \ldots, m)$ based on each disjunction D_i in Q_D's WHERE clause (Lines 3–6). The WHERE clause of each Q_i contains only one conjunction in the canonical form, `Ai1 AND ... AND Ain`. We apply Algorithm 1 to generate its corresponding auxiliary query $\tilde{Q}_i$ (Line 7) and apply Algorithm 2 to generate program input values R_i (Line 8). The union of R_is then contains all appropriate program input values (Line 9). Note that dealing with conjunction of disjunctions could be exponential to the size of disjunctions, causing much more cost when running the generated auxiliary queries against the existing database.

4.2 Query with group-by and aggregation

The query with group-by and aggregation has the form:

```
SELECT  C1, ..., Ch, AGG1(B1), ..., AGGf(Bf)
FROM    from-list
WHERE   A1 AND A2 ... AND An
GROUP BY G1, ..., Gg
HAVING group-qualification
```

The WHERE clause of the query is a conjunction of simple predicates. `AGG1, ..., AGGf` represent built-in SQL aggregate functions (e.g., Sum, Max, or Min) and are computed over the groups. We refer to columns `B1, ..., Bf` as the aggregating columns and columns `G1, ..., Gg` as grouping columns of the query. SQL semantics require that attributes `C1, ..., Ch` in `select-list` must be among `G1, ..., Gg` in `grouping-list`. The expressions appearing in the group-qualification in the

```
01:public int calcStat(int type,int zip) {
     ...
11:  string query = "SELECT C.age, AVG(C.income) as avg1,"
    +" AVG(M.balance) as avg2 FROM customer C, mortgage M"
    +" WHERE M.year='" + years +"' AND"
    +" C.zipcode='"+ fzip + "' AND C.SSN = M.SSN"
    +" GROUP BY C.age"
    +" Having COUNT(*) > 2";
     ...
14:  while (results.Read()){
15:     int age = int.Parse(results["age"]);
16:     int income = int.Parse(results["avg1"]);
17:     int balance = int.Parse(results["avg2"]);
18:     int diff = income - 1.5 * balance;
19:     if (diff > 100000){
20:        ...;}}
21:  return;}
```

Fig. 3 An example code with a group-by query

optional HAVING clause must have a single value per group. A column appearing the `group-qualification` must appear as the argument to an aggregation operator, or it must also appear in `grouping-list`. If GROUP BY is omitted, the entire returned result is regarded as one single group.

Consider the modified example code in Fig. 3 where we have a query with group-by and aggregation (Line 11) and a branch condition involving the returned aggregate values (Line 18). With input "`type=0, zip=0`", in Line 11, we dynamically get the concrete query string as

```
Q:  SELECT C.age, AVG(C.income), AVG(M.balance)
    FROM customer C, mortgage M
    WHERE M.year=15 AND C.zipcode=1 AND C.SSN=M.SSN
    GROUP BY C.age
    HAVING count(*)>2
```

Through static analysis, we also get `Q`'s corresponding abstract form as

```
Qabs: SELECT C.age, AVG(C.income), AVG(M.balance)
      FROM customer C, mortgage M
      WHERE M.year=:years AND C.zipcode=fzip AND C.SSN=M.SSN
      GROUP BY C.age
      HAVING count(*)>2
```

The execution of `Q` on Table 2 yields zero records. Thus, the `while` loop body in Lines 14–20 is not entered.

Algorithm 4 shows our pseudo procedure to construct the auxiliary query given a query with group-by and aggregation. Similar to Algorithm 1, we also have three cases: predicate expressions in the WHERE clause are data-dependent on program inputs, program variables in branch conditions after executing the query are data-dependent on the query's result set, and program variables in branch conditions after executing the query are data-dependent on program inputs. In Algorithm 4, we use bold fonts to emphasize those steps that are different from Algorithm 1.

Algorithm 4 Auxiliary Query Construction for Query with Group-by and Aggregation

Input: **a group-by query** Q, Q's abstract form Q_{abs},
program input set $I = \{I_1, I_2, ..., I_k\}$,
path condition $PC = pc_1 \wedge pc_2 \wedge ... \wedge pc_s$
Output: an auxiliary query $\tilde{Q}$

1: Find variables $V = \{V_1, V_2, ..., V_t\}$ data-dependent on I;
2: Decompose Q_{abs} with a SQL parser to **get SELECT, FROM, WHERE, GROUP BY and HAVING clauses**;
3: Construct a predicate set $A = \{A_1, A_2, ..., A_n\}$ from Q's WHERE clause;
4: Construct an empty predicate set $\tilde{A}$, an empty attribute set C_V, and **an empty group-by query** $\tilde{Q}$;
5: **for** each predicate $A_i \in A$ **do**
6: **if** A_i does not contain program variables **then**
7: Leave A_i unmodified and check the next predicate;
8: **else**
9: **if** A_i does not contain program variables from V **then**
10: Substitute A_i's program variables with their corresponding concrete values in Q;
11: **else**
12: Substitute the variables from V with the expression expressed by I;
13: Substitute the variables not from V with their corresponding concrete values in Q;
14: Copy A_i to $\tilde{A}$;
15: Add A_i's associated database attributes to C_V;
16: **end if**
17: **end if**
18: **end for**
19: Append C_V to $\tilde{Q}$'s SELECT clause;
20: Copy Q's FROM clause to $\tilde{Q}$'s FROM clause;
21: Append $A - \tilde{A}$ to $\tilde{Q}$'s WHERE clause, **append Q's grouping-list and C_V to $\tilde{Q}$'s GROUP BY clause, copy Q's HAVING clause to $\tilde{Q}$'s HAVING clause**;
22: Find variables $U = \{U_1, U_2, ..., U_u\}$ coming directly from Q's result set;
23: Find U's corresponding database attributes $C_U = \{C_{U1}, C_{U2}, ..., C_{Uw}\}$;
24: **for** each branch condition $pc_i \in PC$ after Q's execution **do**
25: **if** pc_i contains variables data-dependent on U **then**
26: **if** pc_i does not contain variables data-dependent on V **then**
27: Substitute the variables in pc_i with the expression expressed by the variables from U;
28: Substitute the variables from U in pc_i with U's corresponding database attributes in C_U;
29: Add the branch condition in pc_i to $\tilde{PC}$;
30: **else**
31: Leave pc_i unchanged and check the next branch condition;
32: **end if**
33: **end if**
34: **end for**
35: Append all the branch conditions in $\tilde{PC}$ to $\tilde{Q}$'s **HAVING clause**;
36: **return** $\tilde{Q}$;

Lines 1–21 in Algorithm 4 deal with the case where predicate expressions in the WHERE clause are data-dependent on program inputs. In Line 2, we decompose Q_{abs} with a SQL parser to get SELECT, FROM, WHERE, GROUP BY, and HAVING clauses. Note that the HAVING clause may be empty if the group-by query does not have group qualification. The GROUP BY clause may be empty too, which corresponds to a simple query with aggregation in the SELECT clause. Lines 5–18 examine the relationship between predicates in the WHERE clause and program variables data-dependent on program inputs. The derived database attributes in C_V, which are data-dependent on program inputs, are appended to the SELECT clause of the con-

structed auxiliary query $\tilde{Q}$ (Line 19). In Line 21, the predicates in $A - \tilde{A}$ form the predicates in the $\tilde{Q}$'s WHERE clause. The `grouping-list` in $\tilde{Q}$'s GROUP BY clause contains the attributes in Q's `grouping-list` and the attributes in C_V. The HAVING clause of $\tilde{Q}$ is the same as that of Q. The constructed auxiliary query for the example code (before entering the while loop in Line 14) in Fig. 3 is:

```
SELECT   C.zipcode
FROM     customer C, mortgage M
WHERE    M.year = 15 AND C.SSN = M.SSN
GROUP BY C.age, C.zipcode
Having   COUNT(*) > 2
```

Lines 22–35 in Algorithm 4 deal with the last two cases: program variables in branch conditions after executing the query are data-dependent on the query's result set, and program variables in branch conditions after executing the query are data-dependent on program inputs. The derived branch conditions $\tilde{PC}$ that are data-dependent on the returned result set are appended to the $\tilde{Q}$'s HAVING clause (Line 35 in Algorithm 4) rather than the $\tilde{Q}$'s SELECT clause in Algorithm 1. This is because the branch conditions $\tilde{PC}$ involve program variables that are data-dependent on the returned values of `AGG1(B1), ..., AGGf(Bf)` in the Q's SELECT clause. The constructed auxiliary query for the example code (in Line 18) in Fig. 3 is:

```
SELECT   C.zipcode
FROM     customer C, mortgage M
WHERE    M.year = 15 AND C.SSN = M.SSN
GROUP BY C.age, C.zipcode
Having   COUNT(*) > 2 AND AVG(C.income)-AVG(M.balance) > 100000
```

When executing the constructed auxiliary query against the existing database state, we get the returned `zipcode` values. Executing the program-input generation (Algorithm 2) will return the final program input values for `zip`.

5 Evaluation

Our approach can provide assistance to DSE-based test-generation tools (e.g., Pex Microsoft Research Foundation of Software Engineering Group 2007 for .NET) to improve code coverage in database application testing. In our evaluation, we seek to evaluate the benefit and cost of our approach from the following two perspectives:

RQ1: What is the percentage increase in code coverage by the program inputs generated by Pex with our approach's assistance compared to the program inputs generated without our approach's assistance in testing database applications?

RQ2: What is the cost of our approach's assistance?

We report the code coverage using block coverage measured by Pex. Code coverage could be quantified by code blocks, lines of code, and partial lines if they are executed by a test run. Among these code entities under coverage measurement, a code block is a code path with a single entry point, a single exit point, and a set of instructions that are all run in sequence. In this evaluation, we choose block coverage (e.g., the percentage of code blocks being covered) as the metric. The reasons are twofold. First,

Pex uses block coverage in its coverage report. Second, we find for these applications, the number of covered blocks could generally reflect the covered code portions.

In our evaluation, we also record the number of runs and execution time. A run represents one time that one path is explored by Pex using a set of program input values. Because of the large or infinite number of paths in the code under test, Pex uses exploration bounds to make sure that Pex terminates after a reasonable amount of time. For example, the bound TimeOut denotes the number of seconds after which the exploration stops. In our evaluation, we use the default value TimeOut=120s and use "time out" to indicate timeout cases.

5.1 Evaluation setup

We conduct an empirical evaluation on two open source database applications: `RiskIt`[3] and `UnixUsage`.[4] `RiskIt` is an insurance quote application that makes estimation based on users' personal information, such as zipcode and income. It has an existing database containing 13 tables, 57 attributes, and more than 1.2 million records. `UnixUsage` is an application to obtain statistics about how users interact with the Unix systems using different commands. It has a database containing 8 tables, 31 attributes, and more than 0.25 million records. Both applications were written in Java. To test them in the Pex environment, we convert the Java source code into C# code using a tool called Java2CSharpTranslator.[5] The evaluation is conducted on a machine with hardware configuration Intel Pentium 4 CPU 3.0 GHz, 2.0 GB Memory and OS Windows XP SP2. The DBMS used in this evaluation is Microsoft SQL Server 2005. The detailed evaluation subjects and results can be found on our project website.[6]

In the evaluation part of the conference version Pan et al. (2011b) of this paper, we made use of the whole existing databases of the two applications. To better evaluate the effectiveness of our approach, as complementary results, in this paper, we choose subsets of the records from the two existing databases with sampling ratios of 5 and 10 %.

In our evaluation, we implement the steps illustrated in the algorithms of our approach by both manual and automated processes. Inputs to the algorithms are obtained by manual collection and include the intermediate information collected during Pex's execution. Then, we derive expected outputs by following the steps in the algorithms. For each method, we first run Pex without our approach's assistance to generate test inputs. We record their statistics of code coverage, including total program blocks, covered blocks, coverage percentages, and the number of generated test inputs. Pex often fails to generate test inputs to satisfy or cover branch conditions that are data-dependent on the query's execution or its returned result set. The not-covered code portions can be easily observed because they are marked by different colors in Pex's UI. Then, before the places where code statements are not covered, we manually identify the corresponding query-issuing point and related branch conditions. We man-

[3] https://riskitinsurance.svn.sourceforge.net.

[4] http://sourceforge.net/projects/se549unixusage.

[5] http://sourceforge.net/projects/j2cstranslator/.

[6] http://www.sis.uncc.edu/~xwu/DBGen.

Table 4 Evaluation results on RiskIt (10 % sample, TO = time out, TC = no. of generated test inpukts)

No.	Method	Case	Total (blocks)	Covered (blocks)			Runs			Time (s)	
				Pex	Pex+ours	Increase (%)	Pex	Ours	TC	Pex	Ours
1	getAllZipcode	1	39	17	37	51.28	12	3	3	TO	4.4
2	filterOccupation	1	41	27	37	24.39	18	4	4	TO	5.6
3	filterZipcode	1	42	28	38	23.81	76	4	4	13.6	4.9
4	filterEducation	1	41	27	37	24.39	76	4	4	TO	5.1
5	filterMaritalStatus	1	41	27	37	24.39	18	4	4	16.3	5.3
6	findTopIndustry Code	1	19	13	14	5.26	32	4	4	TO	5.3
7	findTopOccupa tionCode	1	19	13	14	5.26	81	5	5	TO	4.5
8	updatestability	2	79	61	75	17.72	95	6	6	TO	4.6
9	userinformation	2	61	40	57	27.87	37	3	3	21.2	4.2
10	updatetable	1	60	42	56	23.33	42	3	3	26.4	4.1
11	updatewagetable	3	52	42	48	11.54	75	8	6	TO	5.1
12	filterEstimated Income	1	58	44	54	17.24	105	8	6	TO	4.5
13	calculateUnemploy mentRate	1	49	45	45	0.00	89	7	5	TO	4.6
14	calculateScore	1	93	16	87	76.35	92	10	7	TO	4.4
15	getValues	1	107	38	99	57.01	182	43	9	TO	6.7
16	getOneZip code	1	34	23	32	26.47	22	6	6	TO	6.1
17	browseUser Properties	1	108	85	104	17.60	83	9	7	TO	10.3
All methods (total)			943	588	871	25.52	1135	131	86	1637.5	89.7

ually instrument code statements by calling Pex API methods to get the intermediate information (e.g., path conditions) collected by Pex's exploration. Based on the information required by the aforementioned algorithm on auxiliary query construction, for each program under test, we then perform our algorithm to construct auxiliary queries. We use ZQL as the SQL parser to get structures from the program queries and incorporate the intermediate information collected by Pex to form auxiliary queries. We form auxiliary queries following the aforementioned algorithm on auxiliary query construction. We then execute the auxiliary queries against the existing database and generate new test inputs. For the program-input generation, we use Z3 developed at Microsoft as the constraint solver. We create test inputs in forms that can be executed by Pex using the generated input values. We then run the test inputs previously generated by Pex and the new test inputs generated by our approach, and then record new statistics.

5.2 Code coverage

We show the evaluation results in Tables 4 and 5. For each table, the first part (Columns 1–3) shows the index, method names, and the category (three cases as aforementioned)

Table 5 Evaluation results on UnixUsage (10 % sample, TO = time out, TC = no. of generated test inputs)

No.	Method	Case	Total (blocks)	Covered (blocks)			Runs			time (s)	
				Pex	Pex+ours	Increase (%)	Pex	Ours	TC	Pex	Ours
1	courseNameExists	1	7	6	7	14.29	17	3	3	13.1	5.6
2	getCourseIDByName	1	10	6	10	40.00	14	3	3	12.0	6.0
3	computeFileTo NetworkRatio ForCourseAndSessions	2	25	8	25	68.00	35	7	6	TO	7.9
4	outputUserName	1	14	9	14	35.71	18	4	4	17.4	5.5
5	deptNameExists	1	13	9	13	30.77	18	3	3	22.6	4.8
6	computeBeforeAfter RatioByDept	2	24	8	24	66.67	109	8	7	31.2	7.1
7	getDepartmentIDByName	1	11	7	11	36.36	92	3	3	TO	5.9
8	computeFileToNet workRatioForDept	2	21	20	21	4.76	33	6	5	TO	6.3
9	officeNameExists	1	11	7	11	36.36	18	3	3	23.7	5.8
10	getOfficeIdByName	1	9	5	9	44.44	18	3	3	29.6	5.6
11	raceExists	1	11	7	11	36.36	18	3	3	16.6	5.9
12	userIdExists(version1)	1	11	7	11	36.36	18	3	3	21.3	6.1
13	transcriptExist	1	11	7	11	36.36	18	3	3	18.3	6.3
14	getTranscript	1	6	5	6	16.67	14	2	2	17.2	6.2
15	commandExists(version1)	1	10	6	10	40.00	14	2	2	17.0	6.1
16	categoryExists	1	11	7	11	36.36	18	3	3	15.2	5.5
17	getCategoryByCommand	1	8	5	8	37.50	17	2	2	16.1	5.5
18	getCommandsByCategory	1	10	6	10	40.00	17	2	2	16.4	5.7
19	getUnixCommand	1	6	5	6	16.67	17	2	2	21.5	5.3
20	retrieveUsageHistoriesById	1	21	7	21	66.67	86	3	3	28.7	5.1
21	userIdExists(version2)	1	11	7	11	36.36	19	3	3	14.4	5.2
22	commandExists(version2)	1	11	7	11	36.36	21	3	3	16.2	4.8
23	retrieveMaxLineNo	1	10	7	10	30.00	53	3	3	TO	6.6
24	retrieveMaxSequenceNo	1	10	7	10	30.00	35	3	3	TO	4.7
25	getSharedCommand Category	1	11	7	11	36.36	118	3	3	TO	4.9
26	getUserInfoBy	3	47	15	47	68.09	153	4	3	TO	5.4
27	doesUserIdExist	1	10	9	10	10.00	74	2	2	22.3	5.3
28	getPrinterUsage	3	34	27	34	20.59	115	4	3	39.9	5.8
All methods (total)			394	258	394	34.52	1197	93	89	1270.7	215.8

each method belongs to. The second part (Columns 4–7) shows the code coverage result. Column 4 "total(blocks)" shows the total number of blocks in each method. Columns 5–7 "covered(blocks)" show the number of covered blocks by Pex without our approach's assistance, the number of covered blocks by Pex together with our approach's assistance, and the percentage increase, respectively.

Within the `RiskIt` application, 17 methods are found to contain program inputs related with database attributes. Among them, 14 methods belong to Case 1, 2 methods belong to Case 2, and 1 method belongs to Case 3. In addition, 11 methods contain simple SELECT queries, 3 methods contain DPNF queries, and 3 methods contain queries with group-by and aggregation. These 17 methods contain 943 code blocks in total. Test inputs generated by Pex without our approach's assistance cover 588 blocks while Pex with our approach's assistance covers 871 blocks. In fact, Pex with our approach's assistance can cover all branches except those branches related to exception handling.

The `UnixUsage` application contains 28 methods whose program inputs are related with database attributes, with 394 code blocks in total. Among 28 methods, 23 methods belong to Case 1, 3 methods belong to Case 2, and 2 methods belong Case 3. In addition, 25 methods contain simple SELECT queries, 2 methods contain DPNF queries, and 1 method contains a query with group-by and aggregation. Pex without our approach's assistance covers 258 blocks while Pex with our approach's assistance covers all 394 blocks. The `UnixUsage` application constructs a connection with the database in a separate class that none of these 28 methods belong to. Thus, failing to generate inputs that can cause runtime database connection exceptions has not been reflected when testing these 28 methods.

Note that the code coverage reported in Tables 4 and 5 is the same as the result of the conference version Pan et al. (2011b) of this paper. The reason is that, for both sampling ratios 10 and 5 %, the sizes of sampled subsets are still large, using which Pex could achieve code coverage as high as it could try while our approach could help get effective additional program input values for achieving higher code coverage than Pex.

We pick method No. 1 `getAllZipcode` of `RiskIt` as a basic example to illustrate the evaluation details. Method `getAllZipcode` takes a `String` type variable `zipcode` as parameter and combines it into an SQL query. The domain for `zipcode` is too large for Pex to choose an appropriate value. Hence, the `while` loop cannot be entered by Pex without our approach's assistance. After running the constructed auxiliary query against the existing database state, with the aforementioned two sampling ratios, we get 158 distinct zipcode values for the 10 % sample and 81 for the 5 % sample. As the program input, any one of these 158 or 81 values can be used as the program input in order to cover the `while` loop in Line 08. The method contains 39 blocks in total. Pex without our approach's assistance covers 17 blocks while Pex with our approach's assistance covers 37 blocks. The two not-covered blocks belong to the `catch` statements, which mainly deal with program exceptions at runtime.

Method No. 12 `filterEstimatedIncome` of `RiskIt` has a program input parameter `String getIncome` that is involved in a mathematical formula comparing with the SQL query's returned record values. We use the constraint solver Z3 to derive appropriate program input values. We notice that for method No. 13, our approach covers the same number of blocks as Pex does. This method takes one state name as input, combines it into a query string, queries the total number of the unemployed people living in this state, and calculates the unemployment rate. Pex without our approach's assistance generates a random string value for state names. The number of records returned is always zero, which can still incur the program execution. However, Pex

with our approach's assistance can generate meaningful state names such as "NC" and meaningful records can be returned during the testing.

Within the `RiskIt` and `UnixUsage` applications, only one method, method No. 14 of `RiskIt`, incurs data instantiation. For this method, we need to generate and insert new records into the database because the existing database state does not contain any record that satisfies the executed query's conditions. For example, in method No. 14 `calculateScore`, one branch condition checks whether the returned result's attribute `occupation` is equal to `Federal government`. Unfortunately, among the existing 1.2 million records, there are no such record including this attribute value. Thus, our approach detects this issue and inserts corresponding auxiliary records into the database. We notice that methods No. 13 and No. 14 involve aggregate calculations at the code level. Following aforementioned algorithm dealing with aggregate calculations, we construct GROUP BY queries to determine appropriate inputs for these two methods.

Within the `RiskIt` and `UnixUsage` applications, there is only one method, method No. 15, which contains multiple queries (four queries). The method accepts an input `SSN` and constructs various queries on different tables. We observe that all four queries are independent with each other. We construct one auxiliary query for each original query and derive effective values for the input `SSN`.

For methods No. 16 and No. 17 of `RiskIt`, their parameters have complex abstract data types that are predefined in other classes. For example, the data type of method `getOneZipcode`'s parameter is `Invitation`. One field `SSN` of the object `Invitation` appears in the query. Pex is able to generate an object with the type `Invitation` as input; however, Pex cannot associate the field `SSN` with the database attribute `SSN`. Our approach constructs the auxiliary query, retrieves meaningful `SSN` values from the database, and then builds new meaningful object with the data type `Invitation` as the program input.

Three methods from the `RiskIt` and one method from the `UnixUsage` contain queries with group-by and aggregation. For example, method No. 6 of `UnixUsage` contains one group-by query that calculates summations of system usage for two separate sessions by considering each department as one group. Its program input parameter `semester` is incorporated in the WHERE clause of the embedded group-by query. The result set is iterated for calculating the system usage ratio for each department. Following the aforementioned algorithm that deals with group-by and aggregation, we form an auxiliary query based on the original query by modifying the GROUP BY clause using the `semester` attribute. Executing the auxiliary query will then return a set of string values for qualified semesters.

5.3 Cost

In Tables 4 and 5, the third part (Columns 8–12) shows the cost. Column 10 shows the number of test inputs generated by our approach. Columns 8 and 11 "Pex" show the number of runs and the execution time used by Pex without our approach's assistance. We notice that, for both applications, Pex often terminates with "time out". The reason is that Pex often fails to enter the loops of iterating the returned result records. Columns 9 and 12 "ours" show the additional number of runs by Pex with assistance of our

approach and the extra execution time (i.e., the time of constructing auxiliary queries, deriving program input values by executing auxiliary queries against the existing database, and running new test inputs) incurred by our approach. For the execution time used by Pex and extra execution time incurred by our approach, due to space limit, we only list the results for the 10 % sample in the tables and we summarize the results for the 5 % sample. The number of runs required by Pex and our approach and the number of generated test inputs by our approach are the same as the 10 % sample, as under both sampling ratios, we need the same number of runs to collect information required by our approach and correspondingly generate the same number of test inputs to cover desired program code. For `RiskIt`, Pex uses 1598.7 s in total for all methods while our approach uses additional 51.3 s in total. For `UnixUsage`, Pex uses 1066.3 s in total for all methods while our approach uses additional 124.7 s in total. For the 5 % sample, both Pex and our approach use less time than the 10 % sample. The reason is that running queries against databases with smaller sizes costs less time.

We observe that the most parts of the execution time of both Pex and our approach come from the queries' execution against the database as the associated two databases contain large numbers of records. Although we conduct samplings from the records, the sizes of the samples are still large.

The results show that, for both applications, Pex with assistance of our approach achieves much higher code coverage with relatively low additional cost of a few runs and a small amount of extra execution time. In our evaluation, we set the TimeOut as 120 s. For those "time out" methods, Pex could not achieve new code coverage even given larger TimeOut values. Our approach could effectively help cover new branches not covered by Pex with relatively low cost.

Note that in our current evaluation, we loosely integrate Pex and our approach: we perform our algorithms only after Pex finishes its previous exploration (i.e., after applying Pex without our approach's assistance) since our algorithms rely on the intermediate information collected during Pex's exploration. We expect that after our approach is tightly integrated into Pex, our approach can effectively reduce the overall cost of Pex integrated with our approach (which is currently the sum of the time in Columns 9 and 10). In such tight integration, our algorithms can be triggered automatically when Pex fails to generate test inputs to satisfy branch conditions that are data-dependent on a query's execution or its returned result set.

5.4 Threats to validity

The threats to external validity primarily include the degree to which the subject programs and existing database states are representative of true practice. The studied two open source database applications are of medium size. The studied existing database states have 1.2 million records and 0.25 million records, respectively. These threats could be reduced by more experiments on wider types of subjects and existing database states in future work. The threats to internal validity are instrumentation effects that can bias our results. Faults in our prototype, manual efforts for some not-yet-automated parts, and the underlying Pex and ZQL tools might cause such effects. To reduce these threats, we manually inspected sampled runtime traces of our approach for each program subject. One threat to construct validity is that our evaluation makes

use of the block coverage as the measurement of the effectiveness of database application testing. There exist some coverage metrics (Kapfhammer and Soffa 2003) specific for database application testing, and we plan to investigate such coverage metrics and compare them with the block coverage in further future studies.

6 Related work

Database application testing has attracted much attention recently (Chays 2004; Chays et al. 2004, 2008; Deng and Chays 2005; Kapfhammer and Soffa 2003; Zhou and Frankl 2011a,b, 2012; Emmi et al. 2007; Li and Csallner 2010; Wu et al. 2005, 2007). The AGENDA project (Chays 2004; Chays et al. 2004; Deng and Chays 2005) addressed how to generate test inputs to satisfy basic database integrity constraints. In Chays et al. (2008), parametric queries and constraints on query results during input generation were addressed. The AGENDA database testing toolkit can automatically generate a database state given information about the schema, some generation functions for individual attributes and some user-selected heuristics. The tool also generates test inputs from a simple analysis of the program being verified. The user must then add to each test input preconditions that are checked just before it is executed and that will prevent a case from being executed against an inappropriate database state. Since only one database state is created per test suite, this problem of failed test is likely to become more severe as the size of the test suite grows. An issue with AGENDA is that it cannot guarantee that executing the test query on the generated database states can produce the desired query results. For performance testing, the PPGen prototype system was developed by Wu et al. (2005) and Wu et al. (2007) to generate mock databases by reproducing the statistical distribution of realistic database states. However PPGen assumes constraints are explicit and focus on SQL workload's performance testing.

There is a potential inefficiency involved in generating test descriptions and inputs. The approach of Kapfhammer and Soffa (2003) included within the test description a full specification of the database state against which is to be run and of the database state that should be produced if the test has executed successfully. For example, each DBUnit test is accompanied by an XML file describing the data set required for the test. Before each test run, DBUnit clears the database state and inserts the data described by the XML file. However, it is inefficient since the database must be continually destroyed and recreated between tests, even when significant parts of the database could have been reused by the succeeding tests.

DSE has been explored in database application testing (Willmor and Embury 2006a,b; Emmi et al. 2007; Li and Csallner 2010; Pan et al. 2011b; Taneja et al. 2010). Willmor and Embury (2006a) developed an approach that builds a database state for each test case intensionally, in which the user provides a query that specifies the pre- and post-conditions for the test case. Symbolic execution was used to generate input and database state that will satisfy certain coverage requirements or certain requirements on the intermediate or final results of queries. Emmi et al. (2007) developed an approach for automatic test generation for a database application. The approach was based on DSE and used symbolic constraints in conjunction with a constraint solver to generate both program inputs and database states. In Pan et al. (2011b) and this

extended version, we were focused on program-input generation given an existing database state, avoiding the high overhead of generating new database states during test generation. Pan et al. (2014) recently proposed the SynDB approach that can automatically generate both program inputs and database records by comprehensively incorporating multiple constraints from both program code and database states. The technique is also based on DSE. Li and Csallner (2010) considered a similar scenario, i.e., how to exploit existing databases to maximize the coverage under DSE. However, their approach constructs a new query by analyzing the current query, the result tuples, the covered and not-covered paths, and the satisfied and unsatisfied branch conditions. It can neither capture the close relationship between program inputs and results of SQL queries, nor generate program inputs to maximize code coverage. Recent work of Taneja et al. (2010) proposed an approach called MODA by replacing a real database with mock object. Normal executions like calls to a database are replaced with calls to itself. Symbolic execution on the mock database performs the same result as concrete execution. In this way, it avoids interactions with real database.

Other than choosing high code coverage as the main goal, test-generation for database applications also has other requirements. Halfond and Orso (2006) presented a set of testing criteria called *command form coverage*. It is claimed that all command forms should be covered if issued to the associated database. We developed an approach from Pan et al. (2011a) that leverages DSE to generate database states to achieve advanced structural coverage criteria such as boundary value coverage and logical coverage. The authors in Kapfhammer and Soffa (2008) described a coverage monitoring method for database applications.

Testing of SQL statements has also attracted attention recently (Tuya et al. 2007; Gupta et al. 2010; de la Riva et al. 2010; Tuya et al. 2010). de la Riva et al. (2010) defines a SQL coverage criterion called SQLFpc based on the modified condition decision coverage. In Tuya et al. (2006), Tuya et al. developed a tool to automatically generate mutants of SQL queries. In Tuya et al. (2010), Cabal and Riva developed a method of generating records to satisfy the criterion developed in de la Riva et al. (2010). In Gupta et al. (2010) and Shah et al. (2011), generating test data for killing SQL mutants was investigated. However, these approaches do not consider program constraints and cannot deal with database applications directly. Zhou and Frankl (2009) proposed the JDAMA approach that conducts mutation testing for database applications. The approach is developed to evaluate the performance of mutant-killing based on some given database states.

Testing of database management systems (DBMS) has also attracted much attention (Binnig et al. 2007a,b; Khalek et al. 2008; Bruno et al. 2005). Bruno et al. (2005) introduced a specification language called DGL to generate databases with complex synthetic distributions and inter-table correlations. However, the method does not take queries into consideration while generating the data. Hence, executing the test queries might not return meaningful results. Khalek et al. (2008) proposed a black-box testing approach using the Alloy toolset and developed an ADUSA prototype to generate database states and expected results of executing given queries on generated database states, given a database schema and an SQL query as input. Binnig's papers, Binnig et al. (2007a,b), used symbolic execution and were based on symbolic query processing to generate query-aware databases. These approaches used the information from the

queries to constrain the data generator to generate query-aware databases. Specifically, the generated database can guarantee the size of the intermediate join results to test the accuracy of the cardinality-estimation components or guarantee the input and the output sizes for an aggregation operator in order to evaluate the performance of the aggregation algorithm. In Binnig et al. (2008), Binnig et al. developed a technique called MRQP which can be used to specify and generate test databases for OLTP applications. The technique can help a tester specify the test database for each test case individually using declarative test database specification language. The database generator then creates a test database for a given database schema which returns the given expected result for each query of the specification. The authors Khalek and Khurshid (2010) focused on the automated generation of queries instead of only the data.

SQL queries could involve nested subqueries with aggregation functions, union/union-all, distinct, and group-by views, etc. A large body of work exists on query transformation in databases. Various decorrelation techniques (e.g., Kim 1982; Dayal 1987) have been explored to unnest complex queries into equivalent single level canonical queries. Kim Kim (1982) first showed four basic types of nesting, *Type-A, Type-N, Type-J and Type-JA*, and developed query transformation algorithms to rewrite nested queries into equivalent, flat queries. Dayal (1987) refined and extended all of the previous optimzation work to a unified approach for processing queries that contain nested subqueries, aggregates and quantifiers, which enable unnesting queries with more than one nesting level. Queries with non-aggregated subqueries can be unnested as shown in (Kim 1982; Dayal 1987; Ganski et al. 1987). Unnesting of aggregate subqueries has been studied extensively in (Gupta et al. 1995; Chaudhuri and Shim 1994). Recent work of Ahmed et al. (2006) showed that almost all types of subqueries can be unnested except those that are correlated to non-parents, whose correlations appear in disjunction, or some ALL subqueries with multi-item connecting condition containing null-valued columns. However, when transforming correlated subqueries into canonical forms, viewed tables or derived tables are introduced, which incur challenges in auxiliary query reconstruction. Note that viewed tables and derived tables are used temporarily only for query optimization/execution and do not exist in physical databases.

7 Conclusion and future work

In this paper, we have presented an approach that takes database applications and a given database as input, and generates appropriate program input values to achieve high code coverage. In our approach, we employ DSE to analyze the code under test and formulate auxiliary queries based on extracted constraints to generate program input values. We incorporate a data-instantiation component in our framework to deal with the case that no effective program input values can be attained. We determine how to generate new records and populate them in the new database state. We also extend our approach of program-input generation to database applications including canonical queries and group-by queries. Empirical evaluations on two open source database applications showed that our approach can assist Pex, a state-of-the-art DSE tool, to generate program inputs that achieve higher code coverage than the program inputs generated by Pex without our approach's assistance.

In our future work, we plan to extend our technique to construct auxiliary queries directly from embedded complex queries (e.g., nested queries), rather than from their transformed norm forms. Real database applications may involve complex expressions (e.g., transforming program inputs in predicates of SQL statements or iterating returned results after query execution). We will study strategies of dealing with complex expressions in database application testing. Many real database applications include not just single queries, but rather they include multiple queries embedded in the application program. In our future work, we plan to extend our approach on auxiliary query reconstruction to attain effective program input values when multiple queries are embedded. An execution path of an application can also involve the execution of a sequence of SQL statements including both SELECT queries and state-modifying SQL statements such as INSERT,UPDATE and DELETE. We aim to explore how to extend our approach to state-modifying SQL statements. Finally we will conduct empirical evaluations with more real-world applications.

Acknowledgments This work was supported in part by U.S. National Science Foundation under CCF-0915059 for Kai Pan and Xintao Wu, and under CCF-1349666, CNS-1434582, CCF-1434596, CCF-1434590, and CNS-1439481 for Tao Xie.

References

Ahmed, R., Lee, A.W., Witkowski, A., Das, D., Su, H., Zaït, M., Cruanes, T.: Cost-based query transformation in Oracle. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 1026–1036 (2006)

Binnig, C., Kossmann, D., Lo, E.: Reverse query processing. in: Proceedings of IEEE International Conference on Data Engineering (ICDE), pp. 506–515 (2007a)

Binnig, C., Kossmann, D., Lo, E., Özsu, M.T.: QAGen: generating query-aware test databases. In: Proceedings of ACM SIGMOD Conference pp. 341–352 (2007b)

Binnig, C., Kossmann, D., Lo, E.: Multi-RQP: generating test databases for the functional testing of OLTP applications. in: International Workshop on Testing Database Systems (DBTest), p. 5 (2008)

Bruno, N., Chaudhuri, S.: Flexible database generators. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 1097–1107 (2005)

Chaudhuri, S., Shim, K.: Including group-by in query optimization. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 354–366 (1994)

Chays, D.: Test data generation for relational database applications. PhD thesis, Computer and Information Science, Polytechnic University (2004)

Chays, D., Deng, Y., Frankl, P.G., Dan, S., Vokolos, F.I., Weyuker, E.J.: An AGENDA to test relational database applications. J. Softw. Test. Verif. Reliab. **14**, 17–44 (2004)

Chays, D., Shahid, J., Frankl, P.G.: Query-based test generation for database applications. In: International Workshop on Testing Database Systems (DBTest), p. 6 (2008)

Dayal, U.: Of nests and trees: a unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 197–208 (1987)

de la Riva, C., Cabal, M.J.S., Tuya, J.: Constraint-based test database generation for SQL queries. In: International Workshop on Automation of Software Test (AST), pp. 67–74 (2010)

Deng, Y., Chays, D.: Testing database transactions with AGENDA. In: Proceedings of International Conference on Software Engineering (ICSE), pp. 78–87 (2005)

Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA), pp. 151–162 (2007)

Ganski, R.A., Wong, H.K.T.: Optimization of nested SQL queries revisited. In: Proceedings of ACM SIGMOD Conference, pp. 23–33 (1987)

Godefroid, P., Luchaup, D.: Automatic partial loop summarization in dynamic test generation. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA), pp. 23–33 (2011)

Gupta, A., Harinarayan, V., Quass, D.: Aggregate-query processing in data warehousing environments. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 358–369 (1995)
Gupta, B.P., Vira, D., Sudarshan, S.: X-data: generating test data for killing SQL mutants. In: Proceedings of IEEE International Conference on Data Engineering (ICDE), pp. 876–879 (2010)
Halfond, W.G.J., Orso, A.: Command-form coverage for testing database applications. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 69–80 (2006)
Kapfhammer, G.M., Soffa, M.L.: A family of test adequacy criteria for database-driven applications. In: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of software engineering (ESEC/FSE), pp. 98–107 (2003)
Kapfhammer, G.M., Soffa, M.L.: Database-aware test coverage monitoring. In: Proceedings of the 1st India Software Engineering Conference (ISEC), pp. 77–86 (2008)
Khalek, S.A., Khurshid, S.: Automated SQL query generation for systematic testing of database engines. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 329–332 (2010)
Khalek, S.A., Elkarablieh, B., Laleye, Y.O., Khurshid, S.: Query-aware test generation using a relational constraint solver. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 238–247 (2008)
Kim, W.: On optimizing an SQL-like nested query. ACM Trans. Database Syst. **7**(3), 443–469 (1982)
King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
Li, C., Csallner, C.: Dynamic symbolic database application testing. In: Proceedings of International Workshop on Testing Database Systems (DBTest), pp. 01–06 (2010)
Microsoft Research Foundation of Software Engineering Group: Pex: Dynamic Analysis and Test Generation for .NET
Pan, K., Wu, X., Xie, T.: Database state generation via dynamic symbolic execution for coverage criteria. In: Proceedings of International Workshop on Testing Database Systems (DBTest), pp. 01–06 (2011a)
Pan, K., Wu, X., Xie, T.: Generating program inputs for database application testing. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 73–82 (2011b)
Pan, K., Wu, X., Xie, T.: Guided test generation for database applications via synthesized database interactions. In: ACM Transactions on Software Engineering and Methodology, **23**(2), 12 (2014)
Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 263–272 (2005)
Seshadri, P., Pirahesh, H., Leung, T.Y.C.: Complex query decorrelation. In: Proceedings of the Twelfth International Conference on Data Engineering (ICDE), pp. 450–458 (1996)
Shah, S., Sudarshan, S., Kajbaje, S., Patidar, S., Gupta, B.P., Vira, D.: Generating test data for killing SQL mutants: a constraint-based approach. In: Proceedings of the Twelfth International Conference on Data Engineering (ICDE), pp. 1175–1186 (2011)
Taneja, K., Zhang, Y., Xie, T.: MODA: automated test generation for database applications via mock objects. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 289–292 (2010)
Tuya, J., Cabal, M.J.S., de la Riva, C.: SQLMutation: a tool to generate mutants of SQL database queries. In: Proceedings of the Second Workshop on Mutation Analysis, p. 1 (2006)
Tuya, J., Cabal, M.J.S., de la Riva, C.: Mutating database queries. Inf. Softw. Technol. **49**(4), 398–417 (2007)
Tuya, J., Cabal, M.J.S., de la Riva, C.: Full predicate coverage for testing SQL database queries. J. Softw. Test. Verif. Reliab. **20**, 237–288 (2010)
Willmor, D., Embury, S.M.: An intensional approach to the specification of test cases for database applications. In: Proceedings of International Conference on Software Engineering (ICSE), pp. 102–111 (2006a)
Willmor, D., Embury, S.M.: Testing the implementation of business rules using intensional database tests. In: TAIC PART, pp. 115–126 (2006b)
Wu, X., Sanghvi, C., Wang, Y., Zheng, Y.: Privacy aware data generation for testing database applications. In: Proceedings of International Database Engineering and Applications Symposium (IDEAS), pp. 317–326 (2005)
Wu, X., Wang, Y., Guo, S., Zheng, Y.: Privacy preserving database generation for database application testing. Fundam. Inf. **78**(4), 595–612 (2007)

Zhou, C., Frankl, P.G.: Mutation testing for java database applications. In: Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 396–405 (2009)

Zhou, C., Frankl, P.G.: Inferential checking for mutants modifying database states. In: Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 259–268 (2011a)

Zhou, C., Frankl, P.G.: JDAMA: Java database application mutation analyser. Softw. Test. Verif. Reliab. **21**(3), 241–263 (2011b)

Zhou, C., Frankl, P.G.: Empirical studies on test effectiveness for database applications. In: Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 61–70 (2012)