Automated Software Engineering manuscript No.
(will be inserted by the editor)

Incremental Test Data Generation for Database Queries

María José Suárez-Cabal · Claudio de la Riva · Javier Tuya · Raquel Blanco

Received: date / Accepted: date

Abstract Testing database applications is a complex task since it involves designing test databases with meaningful test data in order to reveal faults and, at the same time, with a small size in order to carry out the testing process in an efficient way. This paper presents an automated approach to generating test data (test relational databases and test inputs for query parameters) for a set of SQL queries, with the aim of covering test requirements as obtained from said queries. The test data generation follows an incremental approach where, in each increment, test data are generated to cover a test requirement by re-using test data previously generated for other test requirements. The test data generation for each test requirement is formulated as a constraint satisfaction problem (CSP), where constraints are derived from the test requirement, initial database states and previously generated test data. The generation process is fully automated and supports the execution on complex queries and databases. Evaluation is carried out on a real life application, and the results show that small-size generated test relational databases achieve high coverage scores for the queries under test in a short generating time.

Keywords software testing · database testing · test database generation · test coverage · constraint satisfaction problem (CSP)

María José Suárez-Cabal
University of Oviedo, Department of Computing
Tel.: +34-985-182520
Fax: +34-985-181986
E-mail: cabal@uniovi.es

Claudio de la Riva
E-mail: claudio@uniovi.es

Javier Tuya
E-mail: tuya@uniovi.es

Raquel Blanco
E-mail: rblanco@uniovi.es

1 Introduction

Most business applications rely on relational databases for storing and managing a huge quantity of data spread over multiple tables. These applications interact with the database by means of SQL queries. These queries implement an important part of the business logic of the database application and it is, therefore, a key issue that these queries are validated for correctness.

Among software verification and validation activities, testing is one of the most used in the industry. One type of testing is functional testing, which aims to verify the functionality of the code under test. An important task of the functional testing activities is the design of test data, and the success of testing greatly depends on the quality of the designed test data. In the context of testing SQL queries, the behavior of the query depends not just on the query parameters provided for the current run, but also on the set of rows stored in the database (the database state). Therefore, the design of such test data must consist of the preparation of a test database that covers the different situations that the SQL query can exhibit.

It is common practice that the tester uses a copy of a live (production) database or a database generated by a commercial or academic tool (Houkjær et al 2006) as a test database, executes the query on the test database and then checks that the query gives the desired results. However, it often happens that the query execution returns no rows or returns rows which do not exercise the different behaviors of the query. This is because the test database is not designed taking into account the different situations that the query can exercise and therefore the test database does not contain enough diverse data in each table to be able to reveal possible faults in the query under test. Additionally, comparison between the actual and expected result is difficult when the test database is composed of many tables and rows. A way to avoid these problems is to reduce the production database such that the coverage of the queries is the same for production and reduced databases. This proposal, presented by Tuya et al (2016), searches and finds small representative subsets (reduced databases) that satisfy the same coverage as the initial ones, have similar fault detection ability and make the comparison between actual and expected results easier. Another way is to generate small test databases starting from empty states that is the aim of this work.

Existing test criteria commonly used for testing programs in procedural languages, such as for example branch coverage, have been used to guide the test database design for the query (Khalek et al 2008; Binnig et al 2007a; Veanes et al 2009; Chays et al 2008). However, these criteria are not designed to test particular features of SQL queries that have the different semantics compared to the procedural code, such as the presence of *null* values and JOIN operators. There are some approaches in the literature that focus on the definition of test adequacy criteria specifically for relational databases (Kapfhammer and Soffa 2003; Halfond and Orso 2006; Tuya et al 2007; Suárez-Cabal and Tuya 2009).

Another important issue is related to the cost of the testing process. In database applications, this aspect is critical because the cost of test database preparation can be very high (preparing many data over many joined tables). It is usual that a database application is composed of different SQL queries executing over the same database and therefore it seems natural to use the same test database for testing all the queries, as it reduces the cost of the test preparation and execution. This manual task is not trivial and it is very expensive, and an automated procedure is needed in order to reduce the cost of the test database generation.

The scope of this paper is related to the automation of test relational database generation and we address this challenge (1) by creating meaningful values in the tables that must be diverse enough to be able to reveal faults in a query and (2) by creating databases small enough to keep the testing efficient for multiple queries.

Our approach uses as test criterion the SQLFpc coverage criterion (Tuya et al 2010). This criterion is based on the Modified Condition Decision Coverage (MCDC). Given an SQL query, it provides a set of test requirements, expressed as SQL queries, which the test database must fulfill. Test requirements are derived from decisions in WHERE and HAVING clauses in SQL query, as well as joins, groupings and aggregate functions. The execution of the test requirement against the test database determines whether it is met when the output returns at least a row.

Conceptually, given a set of SQL queries and an initial database state, our goal is to automatically generate a test database that covers all the test requirements by following an incremental approach. Our aim is to start from an initial database state and find a new database state that fulfills as many test requirements as possible. This is done incrementally, by considering one requirement, updating the database state to cover that requirement, then covering another requirement, and so on, until all test requirements have been processed. Thus, the final database state is a test database. For the automatic generation of the database state in each step, we formulate it as a Constraint Satisfaction Problem (CSP) (Tsang 1993) where the constraints are the test requirements and the previous database state, and then we solve them using a general-purpose constraint solver (Prud'homme et al 2015).

In our previous work (de la Riva et al 2010) we addressed the test database generation with support for a reduced kind of SQL queries and a limited evaluation. This paper goes significantly further and makes the following specific contributions:

- An **incremental approach** to populate test relational databases with support for multiple SQL queries. The incremental strategy is achieved by means of re-using the data from previous database states.
- **Automatic support** of the test data generation by means of the formalization of the incremental approach as a Constraint Satisfaction Problem (CSP).

- Support for a **large set of SQL queries** including SELECT, JOIN, WHERE, GROUP and HAVING clauses.
- **Evaluation** on a real database application with a large number of tables and columns, and using two different sets of SQL queries with different complexity.

The rest of the paper is organized as follows. Sect. 2 introduces the background and notation used in the paper. We present the general approach, the problem and the test database generation algorithm in Sect. 3. The formalization of the database state generation as a CSP is detailed in Sect. 4. In Sect. 5, we present QAGrow tool, which fully automates the approach, and we also present its limitations. Sect. 6 describes the results of the experiments over a real case study and Sect. 7 discusses the related work. The paper ends with conclusions in Sect. 8.

2 Background

2.1 Relational Model Definitions and Notation

In the relational model of databases (Codd 1990), the data in a database is represented as a collection of relations. Here, we give some definitions and we describe the basic notation that will be used in the remainder of the paper.

Relations, tuples and attributes. Given a set of attributes $A = \{A_1, A_2, .., A_n\}$ over a set of domains $D_1, D_2, \ldots, D_n$, a *relation* denoted as $R(A_1, A_2, \ldots, A_n)$ or $R(A)$ is a subset of the Cartesian product of the domains and it is composed of a finite set of m-tuples $R(A) = \{r^1, r^2, \ldots, r^m\}$. A tuple r^j is specified as $r^j =< a_1^j, a_2^j, \ldots, a_n^j >$ where a_i^j is the value of the attribute A_i in the tuple r^j or is a special *null* value which denotes missing information in the value of the attribute. To handle the missing information, we define the Boolean predicate $isnull(a_i^j)$ that is true if the value a_i^j is *null*. In relational database implementations, a relation is a table, a tuple is a row of the table, attributes are the columns of the table and the domains define data types of the attributes. The term *relation schema* refers to the set of attributes of the relation.

Database schema and constraints. A *database schema* S is a set of relation schemas $S = \{R_1, R_2, ..R_n\}$ and a set of constraints specifying restrictions on the database that require relations and attributes to satisfy certain properties. The *nullability constraint* (NOT NULL) forces an attribute not to accept *null* values. We define the predicate $nl(A_i)$ that is true when the attribute A_i is nullable (it has been declared in the database schema without the NOT NULL constraint). The *primary key constraint*, denote as $pk(R)$, specifies a set of attributes in a relation R that uniquely identifies each tuple of the relation. The *foreign key constraint* from a relation R_i to a relation R_j, denoted as $fk(R_i, R_j)$, is a pair of subsets of attributes $\langle A, B\rangle$, where A and B are attributes of R_i and R_j respectively, such that attributes A reference attributes B.

Database states and instances. A database state S refers to the data in a database at a particular point of time. Given a database schema S, a *database state* DS of the schema S is a set of relations $DS = \{R_1, R_2, \ldots, R_n\}$ such as all $R_i s$ satisfy the set of constraints on S. The specific database states managed in this paper are: empty state (the database does not contain data), initial state (the database is loaded with data for the first time) and database instance (the database state loaded with test data).

Database operations and queries . The basic operations over a database are defined as relational assignment in the form $Z \leftarrow rve$, where rve denotes a relation-valued expression (an expression whose evaluation yields a relation) and Z is a relation containing the tuples obtained when applying the rve. In SQL, rve expression is called *query.* Using Codd notation (Codd 1990), the *select operator* $Z \leftarrow R[p(A)]$ (in SQL, `SELECT*FROM R WHERE p(A)`) uses a relation R and generates a relation Z with the tuples of R that satisfy the predicate p on the attributes A. The *inner join operator* $Z \leftarrow R[p(A, B)]S$ (in SQL, `SELECT * FROM R INNER JOIN S ON p(A,B)`) uses two relations, $R(A)$ and $S(B)$, and generates a relation Z with tuples of $R(A)$ concatenated with tuples of $S(B)$ where the logical condition $p(A, B)$ is evaluated to true. The *left outer join* returns the result of the inner join, plus those tuples in $R(A)$ that do not match the join operator. The *right outer join operator* is symmetric to the left join for $S(B)$. The *full outer join* is defined as the union of the inner join, the left outer join and the right outer join. To identify each outer join type in an rve expression, we use the notation $R[p(A, B)]^{JT}S$, where $JT = \{LJ, RJ, FJ\}$ denotes the join type (left, right and full outer join, respectively). The *framing operator* $Z \leftarrow R///G$ divides a relation R into a set of groups where each of them has equal values for a set of attributes G (grouping attributes). The most commonly used is in the form $Z \leftarrow R///G(G, F)$ (in SQL `SELECT G,F FROM R GROUP BY G`) which carries out aggregated calculations performed by aggregate functions (F) over all tuples on each frame. A further select operator may be applied after framing: $Z \leftarrow R///G[q(A, F)](G, F)$ (in SQL, `SELECT G,F FROM R GROUP BY G HAVING q(A,F)`). Predicate $q(A, F)$ involves attributes A in R and aggregate functions over A, and it is called *frame predicate.* In order to clarify these operators, Table 1 summarizes the equivalences between this notation, relational and extended relational algebra notation, and SQL statements.

2.2 SQLFpc Test Coverage Criterion

SQLFpc (Tuya et al 2010) is a test criterion that specifies test requirements (or test situations) specifically tailored to handle the details of SQL queries. It is based on the logical coverage criterion MCDC (Chilenski 2001) that specifies

Table 1 Equivalences between operators, different notations and SQL statements

Operator	**Codd notation**	**Relational algebra, Extended relational algebra**	**SQL statement**
Select	$R[p(A)]$	$\sigma_{p(A)}(R)$	`SELECT * FROM R WHERE p(A)`
Inner join	$R[p(A,B)]S$	$R_{p(A,B)}S$	`SELECT * FROM R INNER JOIN S ON p(A,B)`
Left, right, full outer join	$R[p(A,B)]^{JT}S$, where $JT = \{LJ, RJ, FJ\}$	$R\alpha_{p(A,B)}S$, where $\alpha = \{⟕, ⟖, ⟗\}$	`SELECT * FROM R [LEFT | RIGHT | FULL] OUTER JOIN S ON p(A,B)`
Framing	$R///G$	${}_G\Im(R)$	`SELECT G FROM R GROUP BY G`
Framing and aggregation functions	$R///G(G,F)$	${}_G\Im_F(R)$	`SELECT G,F FROM R GROUP BY G`
Select after framing (frame predicate)	$R///G[q(A,F)](G,F)$	$\sigma_{q(A,F)}({}_G\Im_F(R))$	`SELECT G,F FROM R GROUP BY G HAVING q(A,F)`

test requirements such that every condition in a logical decision has taken all possible outcomes, and each condition has been shown to independently affect the decision's outcome. For example, consider the decision (`A and B`). To satisfy MCDC, for each condition we must generate a pair of test inputs. For the condition `A`, a pair that satisfies the criterion is (1,1) and (0,1), because when the value of condition `A` changes (while the rest of the conditions do not change) the result of the decision changes. For the condition `B`, only the test case (1,0) is generated.

Based on these principles, SQLFpc provides a criterion for SQL queries, where the test input is the database and the programs are the SQL queries. In addition to decisions in WHERE and HAVING clauses, the SQLFpc criterion deals with the way in which SQL queries perform the joins, groupings and aggregations, as well as the handling of the three-valued logic.

Given an SQL query, SQLFpc specifies a set of test requirements in order to fulfill the criterion. These test requirements impose a set of constraints on the database in order to achieve the coverage, which are called *SQLFpc coverage rules* and are expressed as SQL queries. Consider for example, a query `SELECT ID FROM Order INNER JOIN Customer ON customerID = ID WHERE price > 10`. One of its coverage rules is `SELECT * FROM Order INNER JOIN Customer ON customerID = ID WHERE not (price>10)`. The coverage rule is fulfilled if when it is executed against a test database, the output returns at least one row. In this example, the test database must contain rows where the condition in the WHERE clause (`price>10`) is false.

Table 2 Sample database schema and SQL queries

Table Customer		**Table Order**	
Column	**Integrity Constraints**	**Column**	**Integrity Constraints**
ID	PK	orderID	PK
name		customerID	FK, NOT NULL
		price	NOT NULL
		quantity	

Q1:
```
SELECT ID, orderID FROM Order INNER JOIN Customer ON customerID=ID
WHERE quantity>5
```
Q2:
```
SELECT ID FROM Order INNER JOIN Customer ON customerID=ID WHERE
price>10
```

Although the primary use of the SQLFpc criterion is for assessing the coverage of the test data in relation to a query that is executed, it can also be used for designing meaningful test inputs. This is the purpose of this paper.

3 Test Database Generation

This paper addresses the following general problem: Given a set of SQL queries, a database schema and a set of test requirements (specified as SQLFpc coverage rules), find a set of *test database instances* such that the test requirements are fulfilled by them. Due to the fact that there may exist many different database instances that meet these criteria, the goal is to find a small number of them with a reduced size which satisfy the test requirements.

3.1 General Approach and Overview

In order to keep the presentation of the approach concise, we use a simple example of an online store interacting with a database for customers and orders. Table 2 illustrates the database schema and two sample SQL queries issued from a reporting application over the database. Query Q1 lists customers and their orders when the order quantity is greater than 5, and query Q2 selects customers who have orders where the order price is greater than 10.

In order to test these queries adequately, it is necessary to design diverse test data that cover the different situations of each query. We use the SQLFpc test coverage criterion as test criterion for guiding the test database generation (Tuya et al 2010). Informally, to fulfill this criterion for Q1, the test database must include rows with the following test requirements: customers with orders where the condition `quantity>5` is true (Q1.1) and false (Q1.2), the column `quantity` is NULL (Q1.3) and customer without joined orders (Q1.4). Additionally, the test database must meet the integrity constraints of the database schema. Similar test requirements can be derived for Q2: related rows where `price >10` is true (Q2.1), false (Q2.2) and customers without joined orders (Q2.3).

In essence, our approach follows a constraint-solving problem. Given an SQL under test, the SQLFpc test situations define the constraints that the rows in the test database must fulfill. To find these rows, a constraint solver is used to solve the constraints.

A first approach involves generating the test database by means of a *global* solving approach. It starts from a blank test database and then solves *all* the constraints for all queries in a single step. If the solver finds a solution, this solution is the test database. However, if there are two or more test requirements (or integrity constraints) that are inconsistent with each other, the solver returns no solution. A possible solution to the above problem is to solve each test situation individually together with all database constraints (*individual solving*). In this case, the solver is called as many times as test requirements exist in the query and thus a test database is generated for each test situation. But this approach is impractical and it is quite far from the initial goals due to the high number of test databases that could be generated (for testing the Q1 query, we have four different test databases).

We propose an *incremental* solving approach to generate the test database by means of re-using previous generated data (*test database states*) on previous test requirements. Now, we assume for simplicity that the test database is empty to begin with. It starts solving the conditions for (Q1.1) (`quantity>5 and customerID=ID`) and, as a result, a test database state is generated assigning values to the columns in the constraints. Next, the process continues with the test requirement Q1.2. To cover it, the row in the *Customer* table with `ID=1` can be re-used if an order references this customer and it satisfies the condition `not(quantity>5)`. This order is added and the process continues with the remaining test requirements of Q1 and Q2. Finally, values of attributes (*Customer.name*), which do not affect the fulfillment of test requirements, can be randomly generated. Table 3 shows the database states for each test requirement. The final test database state that can be used as test database for the queries Q1 and Q2, with a reduced size (there are no unnecessary rows) and high coverage (the test data considers each different test situation).

3.2 Problem Statement

We enunciate the test database generation problem as follows:

Definition 1.- (*Test database generation problem*). Given a database schema S, an initial database state DS_0, and a set of queries $Q = \{Q_1, Q_2, \dots, Q_n\}$ under test.
Problem: Find a set of database instances $DB = \{DB_1, DB_2, \dots, DB_m\}$ in order to cover each test requirement derived from the queries in Q.

Applying the SQLFpc criterion on each query of Q, a set of coverage rules is obtained. Therefore, the problem is re-formulated as obtaining the set of

Table 3 Incremental test database generation. (a) Detailed process for the test requirements of the query Q1. (b) Final test database state for the queries Q1 y Q2. The values added/updated to the database state to cover the test situation are highlighted in bold.

Test requirements	**Test database states**					
	Customer		**Order**			
	ID	**name**	**orderID**	**quantity**	**price**	**customerID**
(a)						
(Q1.1): set of joined rows where Order quantity is greater than 5	**1**		**1**	**6**		**1**
(Q1.2): set of joined rows where Order quantity is not greater than 5	1		1	6		1
			2	**5**		**1**
(Q1.3): rows such that Order quantity is NULL	1		1	6		1
			2	5		1
			3	**NULL**		**1**
(Q1.4): rows in Customer without joined rows in Order	1		1	6		1
	2		2	5		1
			3	NULL		1
(b)						
All the test requirements for Q1 and Q2	1	**X**	1	6	**11**	1
	2	**Y**	2	5	**10**	1
			3	NULL	**11**	1

database instances DB such that each coverage rule is covered by at least one of them. Our aim is to find the test database instances by means of partial and incremental database states for each coverage rule. We enunciate this problem as follows:

Definition 2.- (*Database state generation problem for a coverage rule*). Given a database state DS_{i-1}, and a coverage rule CR.
Problem: Starting from the database state DS_{i-1}, find a database state DS_i such that CR is covered by DS_i (i.e. when executing CR against DS_i, at least one row is returned).

The approach to find the database state DS_i from DS_{i-1} is based on two basic operations: (1) inserting rows and (2) modifying values in DS_{i-1} such that CR is covered. We denote $T(DS_{i-1}, CR)$ as the function that obtains DS_i, defined as follows:

Definition 3.- (*Transformation function*). Given a database state DS_{i-1}, and a coverage rule CR, $T(DS_{i-1}, CR)$ is the function that obtains DS_i, starting from DS_{i-1}, such that CR returns at least one row.

Fig. 1 illustrates the above definitions.

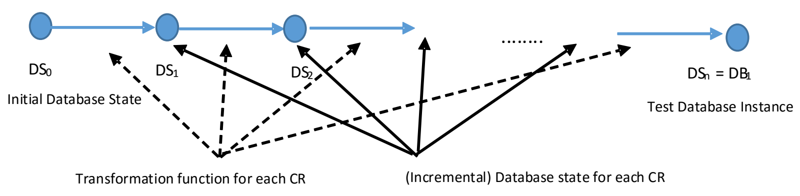
DS0
DS1
DS2
....
DSn = DB1
Initial Database State
Test Database Instance
Transformation function for each CR
(Incremental) Database state for each CR

Fig. 1 Database State and Test Database Instance generation

3.3 General Algorithm

Given a database state DS_{i-1} and a coverage rule CR, a new database state DS_i is obtained by applying a series of changes on DS_{i-1}.Three particular cases can be distinguished for finding solutions to database state generation:

– Case 1: The database state DS_{i-1} already covers the coverage rule CR, then DS_{i-1} is the solution: $DS_i = DS_{i-1}$.
– Case 2: The database state DS_{i-1} does not cover the coverage rule CR. The solution DS_i is found by transforming the database state DS_{i-1} by applying the function T (inserting rows and/or updating values): $DS_i = T(DS_{i-1}, CR)$.
– Case 3: The database state DS_{i-1} does not cover the coverage rule and it is not possible to find any transformation in order to obtain DS_i and cover CR (the coverage rule is inconsistent with data in DS_{i-1}). Then a solution does not exist modifying DS_{i-1}: $DS_i = DS_{i-1}$.

After all coverage rules are processed:

– the final database state will be the database instance that will be added to the final solution (DB), and
– the set of coverage rules not covered ($CRNotCovered$) will be processed again, but starting from an empty database, until there are no rules in $CRNotCoverved$ that can be covered.

This procedure (depicted in Algorithm 1) is performed incrementally in two aspects: 1) using the database state generated for each coverage rule as the initial state for the next rule and 2) repeating the generation of new database instances for those coverage rules not covered before.

To illustrate this procedure, consider a database loaded initially with data (DS_0), and a subset of three coverage rules depicted in Table 4. The database schema has two tables R and S: R has a primary key, attribute ID, and the attributes a and b, and S has the attribute ID as primary key and the attributes c, which is a foreign key referencing $R.ID$, and d.

Table 5 details the incremental procedure for obtaining the solution (a set of test database instances). The Cov.Rule column contains the coverage rule

Algorithm 1 Test database instance generation

```
1: procedure TestDatabaseInstanceGeneration
2: Input: initial database state $DS_0$, a set of queries under test $Q$
3: Output: a set of test database instances ($DB$) that cover the test requirements of $Q$
4:
5:     $CRSet \leftarrow$ set of coverage rules of $Q$ to be covered obtained by applying SQLFpc test coverage criterion
6:     $DS_{i-1} \leftarrow DS_0$
7:     $DB \leftarrow$ empty set
8:     repeat
9:         $CRNotCovered \leftarrow$ empty set
10:        for each coverage rule $CR$ in $CRSet$ do
11:            /*Determine $DS_i$, starting from $DS_{i-1}$, for covering $CR$ (database state generation problem)*/
12:            if $CR$ is covered by $DS_{i-1}$ (case 1) then
13:                $DS_i \leftarrow DS_{i-1}$
14:            else
15:                if there is a transformation, $T$, of $DS_{i-1}$ that covers $CR$ (case 2) then
16:                    $DSi \leftarrow T(DS_{i-1}, CR)$
17:                else
18:                    /* there are no transformations of $DS_{i-1}$ that cover $CR$ (case 3)*/
19:                    $DS_i \leftarrow DS_{i-1}$
20:                    /*Add $CR$ to $CRSetNotCovered$ set to be processed later again*/
21:                    $CRSetNotCovered \leftarrow CRSetNotCovered \bigcup \{CR\}$
22:                end if
23:            end if
24:            /*Next iteration starts from $DS_i$*/
25:            $DS_{i-1} \leftarrow DS_i$
26:        end for
27:        $DB \leftarrow DB \bigcup DS_i$
28:        $CRSet \leftarrow CRSetNotCovered$
29:        $DS_{i-1} \leftarrow$ empty database
30:    until $CRSet$ is empty
31: return $DB$
32: end procedure
```

Table 4 Example of inputs for the test database generation

Database Schema				**Initial DB State(DS_0)**					
R		S		R			S		
ID	PK	ID	PK	ID	a	b	ID	c	d
a		c	FK(ref. R.ID)	1	5	11			
b		d		2	5	10			

Coverage Rules (CR)

CR1. `SELECT * FROM R WHERE a=5`
CR2. `SELECT * FROM R INNER JOIN S ON R.ID = S.c WHERE R.b=10`
CR3. `SELECT * FROM R WHERE ID=1 and a<5`

Table 5 Example of an execution of the test database generation

Cov. Rule	**Case**	**DB State i (DS_i)**						**Rule Ouput**
		R			S			
		ID	a	b	ID	c	d	
CR1	Case 1: Considering DS_0 as DS_{i-1}, this is sufficient for covering $CR1$ and transformations are not necessary $DS_i = DS_{i-1}$	1	5	11				$\{(R.ID, a, b)\} = \{(1, 5, 11), (2, 5, 10)\}$ (2 rows)
		2	5	10				
CR2	Case 2: $CR2$ is covered if the tuple (1,2,) is inserted into S. As the attribute d is irrelevant for $CR2$, it may take any value. It is represented by a blank space.	1	5	11	**1**	**2**		$\{(R.ID, a, b, S.ID, c, d)\} = \{(2, 5, 10, 1, 2,)\}$ (1 row)
		2	5	10				
CR3	Case 3: It is not possible to cover $CR3$ starting from DS_{i-1} because there exists a tuple in R where `ID=1` but `a=5`, and other tuples such that `ID=1` and `a<5` cannot be inserted due to the fact that ID is PK.	No solution						$\{(R.ID, a, b)\} = \{\}$ (no rows)
CR3	Case 2: Starting from an empty database, $CR3$ is covered if a new tuple (1,1,) is inserted into R. Attribute b may take any value, so is represented by a blank space.	**1**	**1**					$\{(R.ID, a, b)\} = \{(1, 1,)\}$ (1 row)

under processing. The Case column indicates what particular case is applied and a brief description of the transformations over DS_{i-1} in order to cover each CR. The results of these transformations allow obtaining a new database state DS_i, shown in the DB State i column where the updates are highlighted in **bold**. Finally, the Rule Output column indicates the output of the coverage rule and the rows returned.

Table 6 Example of an output of the test database generation

Solution (DB) - Test Database Instances											
DB instance DB_1						DB instance DB_2					
R			S			R			S		
ID	a	b	ID	c	d	ID	a	b	ID	c	d
1	5	11	1	2	1	1	1	1			
2	5	10									

The solution is depicted in Table 6, which is composed of two database instances: DB_1 that covers $CR1$ and $CR2$, and DB_2 that covers $CR3$.

Note that at the end of the transformations there may exist attributes whose value does not influence the coverage, and then any value in its domain is assigned, as is the tuple in S (1,2,**1**) in DB_1.

4 Database State Generation as a Constraint Satisfaction Problem

The database state generation problem (**Definition 2**) requires finding the function $T(DS_{i-1}, CR)$ that transforms a database state DS_{i-1} into another DS_i such that a coverage rule CR returns at least one row. Our aim is to generate DS_i determining automatically when a row must be inserted or updated and also the values that must be assigned. To be able to automate these transformations, we state this problem as a *Constraint Satisfaction Problem* (CSP)(Tsang 1993).

In general, a CSP is represented by the tuple $CSP = (X, D, C)$, where X denotes the finite set of variables of the CSP, D the set of domains (one for each variable) and C is a set of constraints. Typically, most constraints can be defined as equalities ($=$) or inequalities ($! =, <, >, \leq, \geq$) of arithmetic expressions over variables, or a boolean combination of such constraints. A *solution* to a CSP is an assignment of values to variables that satisfies all constraints, with each value within the domain of the corresponding variable. A CSP that does not have solutions is called *unsatisfiable.* A *constraint solver* is a tool in charge of finding a solution that satisfies the constraints.

Definition 4.- (*Constraint Satisfaction Problem for database state generation*). Given: a database state DS_{i-1}, and a coverage rule CR. Let a CSP be defined as a tuple (X, D, C) where:

- X is the set of variables $\{X_1, X_2, \ldots, X_n\}$ that represent the database state, the attributes and parameters of the coverage rule;
- D is a function that associates its domain to each variable;
- C is the set of constraints that are defined on a subset of X in order to restrict the possible values for these variables. These constraints are derived from restrictions of the database and the coverage rule.

In general, the solution to a CSP is to find assignments of values from its domain to every variable in X such that every constraint in C is satisfied. In our context, the solution will consist of finding the assignments to the database rows such that the database constraints are satisfied and the coverage rule CR is covered.

Below, the elements of the tuple (X, D, C) and the solution to the CSP are specified for the database state generation problem.

4.1 Variables and Domains

We distinguish two types of variables:

– Stored-Data (SD): represents the database state, initially DS_{i-1}, which is going to be transformed into DS_i when the solution is found. Each variable represents a value in an attribute of a relation, and sets of SD variables may represent tuples of a relation, or even the database depending on the context in which they are used.
– Rule-Output (RO): represents the output tuples of the coverage rule CR. This information must include a row that assures CR is covered, i.e. there is at least one row in the output of CR.

Each variable X_i has a domain depending on the type of the attribute of which it models. Moreover, a particular value is included in each domain: *null* value (represented by NULL) that allows the evaluation of the three-valued logic. Given a variable X_i, the boolean predicate $isnull(X_i)$ is true if X_i has been instanced to NULL and false otherwise.

Each variable X_i has a state in a CSP whose meaning for our problem is:

– Non-instantiated: when its value is not relevant for covering the coverage rule or no value has been assigned yet. Non-instantiated variables will be represented by $\oslash$.
– Instantiated: the variable has been instantiated to a value of the domain that allows covering the coverage rule and it cannot be modified.

Consider the coverage rule `SELECT * FROM R INNER JOIN S ON R.ID = S.c WHERE R.b=10` and the relations $R(ID, a, b)$ and $S(ID, c, d)$ of the previous example (in Table 5). Before the generation, the instantiated variables were only SD, $SD = \{R, S\} = \{\{(1, 5, 11), (2, 5, 10)\}, \{\oslash\}\}$; RO were not instantiated (represented by $\oslash$), $RO = \{R, S\} = \{\{(\oslash, \oslash, \oslash)\}, \{(\oslash, \oslash, \oslash)\}\}$. After the generation, variables are instantiated as $SD = \{\{(1, 5, 11), (2, 5, 10)\}, \{(1, 2, \oslash)\}\}$ and $RO = \{\{(2, 5, 10)\}, \{(1, 2, \oslash)\}\}$. Note that $S.d$ in SD and RO variables continues without instantiating because it is not relevant for covering the rule.

4.2 Constraints

Constraints in the set C are classified in the following categories:

– *database state constraints* (C_{DS}) restrict the values of SD variables to the values of the database state D_{Si-1},

- *schema constraints* (C_S) restrict the values of SD variables in order to fulfill the database schema. Specifically, primary keys, foreign keys and nullability constraints for each relation are considered provided that they are defined. If a primary key or a foreign key in a relation is not defined, the solution will be generated taking into account the rest of constraints established,
- *coverage rule constraints* (C_{CR}) restrict the values of RO variables so that predicates in the coverage rule evaluate to true in order to cover the coverage rule, and
- *state transformation constraints* (C_{ST}) restrict the values of SD and RO variables ensuring each tuple in RO variables exists in SD variables, and transforming the database state DS_{i-1} into the new database state DS_i.

As an illustrative example of the constraint satisfaction process, consider the coverage rule $CR2$ in the previous example (in SQL `SELECT * FROM R INNER JOIN S ON R.ID = S.c WHERE R.b=10`) and the database state DB_{i-1} where $R(ID, a, b) = \{(1, 5, 11), (2, 5, 10)\}$ and $S(ID, c, d) = \{\oslash\}$, shown in Table 5 and Table 6. Initially, the SD and RO variables are non-instantiated. In Fig. 2, the final values in variables are depicted if constraints are satisfied as follows:

1. C_{DS}: the values from DB_{i-1} are assigned to the SD variables.
2. C_{CR}: due to the predicate `R.b=10`, the value 10 is assigned to RO variable $R.b$.
3. C_{ST}: each tuple in RO variables must exist in SD variables. In order to satisfy this constraint, the tuple (2,5,10) in R relation in SD variables may be used for this relation in RO variables because it has the same value in attribute $R.b$. Therefore, the rest of the attributes in SD variables are assigned to the attributes in RO variables.
4. C_{CR}: due to predicate `R.ID=S.c`, the value 2 is assigned to RO variable $S.c$.
5. C_{ST}: there are no tuples in S relation in SD variables then inserting a new tuple for this relation in SD variables is necessary to satisfy C_{ST} constraints. Because at this moment, the only value known in RO variables for S is $S.c$, the new tuple will be $(\oslash, 2, \oslash)$.
6. C_S: the attribute $S.ID$ is PK in S relation, so it must be unique for its tuples. Because there are no values assigned to this attribute in SD variables, the value will be 1.
7. C_{ST}: once again, as in Step 3, the values in the tuple $(1, 2, \oslash)$ in S relation in SD variables, may be assigned to S in RO variables.

Each one of these categories is defined in the following subsections as well as the different types into which they are divided.

4.2.1 Database State Constraints, C_{DS}

Definition 5.- (*Database state constraint*). Given the set of stored-data variables SD and a database state DS_{i-1}, the value of each attribute in each tuple

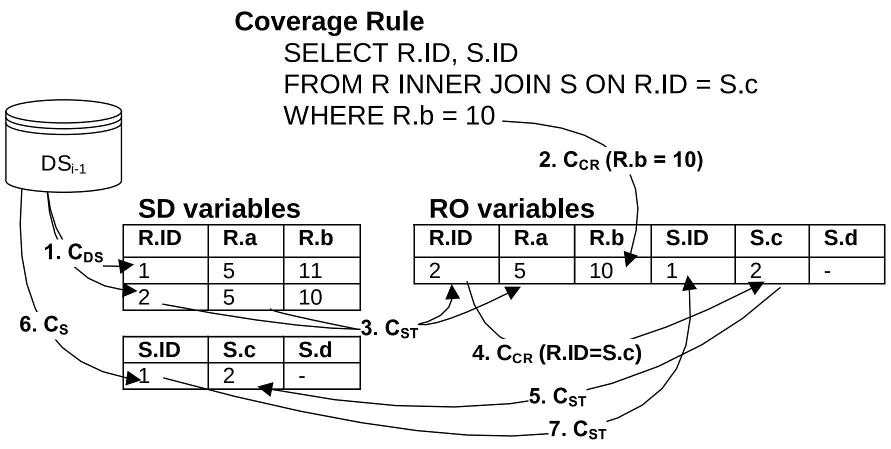

Fig. 2 Solving the problem of generating a database state for a coverage rule starting from DS_{i-1}

of relations in DS_{i-1} is assigned to the corresponding SD variable, remaining as non-instantiated if the value has not been assigned yet in DS_{i-1}. Let C_{DS} be defined as the assignment of the information stored in DS_{i-1} to SD:

$$C_{DS}(SD, DS_{i-1}) := SD \leftarrow DS_{i-1}$$

4.2.2 Schema Constraints, C_S

Definition 6.- (*Primary key constraints*). Given a relation R in SD variables, for each tuple of R, the primary key must be NOT NULL and distinct from the other tuples of R. Let A_i be attributes of R such that $A_i \in pk(R)$, j and k be tuples in R, and a_i^j and a_i^k be the value of A_i in tuples j and k, respectively. The primary key constraint is defined for each A_i as:

$$\begin{aligned} C_S(A_i \in pk(R), j, k) :=& \neg isnull(a_i^j) \wedge \neg isnull(a_i^k) \\ & \wedge (a_i^j = a_i^k \Leftrightarrow j = k) \end{aligned}$$

Definition 7.- (*Foreign key constraints*). Given two relations R and S in SD variables, for which a foreign key is defined in R referencing S. For each attribute, in each tuple, in the foreign key of R, its values are equal to the values of the attribute referenced in any tuple of S or, if the attribute is nullable, they can be NULL. Let A_i be attributes of R and B_i be attributes of S such that $\langle A_i, B_i \rangle \in fk(R, S)$, j and k be tuples in R and S respectively, a_i^j be the value of A_i in tuple j and b_i^k be the value of B_i in tuple k. The foreign key constraint is defined for each A_i and B_i as:

$$\begin{aligned} C_S(\langle A_i, B_i\rangle \in fk(R,S), j, k) := {} & (\neg nl(A_i) \Rightarrow \neg isnull(a_i^j) \wedge \neg isnull(b_i^k) \wedge a_i^j = b_i^k) \\ & \vee (nl(A_i) \Rightarrow isnull(a_i^j) \vee (\neg isnull(a_i^j) \wedge \neg isnull(b_i^k) \\ & \wedge a_i^j = b_i^k)) \end{aligned}$$

Definition 8.- (*Nullability constraints*). Given a relation R in SD variables, for each attribute non-nullable of R, it value in each tuple must be NOT NULL. Let A_i be an attribute of R such that A_i is not nullable, $\neg nl(A_i)$, j be a tuple in R and a_i^j be the value of A_i in tuple j, the nullability constraint is defined as:

$$C_S(A_i, j) := \neg nl(A_i) \Rightarrow \neg isnull(a_i^j)$$

4.2.3 Coverage Rule Constraints, C_{CR}

While the previous constraints are common for all coverage rules under test that are executed against databases with the same schema, coverage rule constraints are specific for each coverage rule restricting the values of RO variables so that its predicates evaluate to true.

Coverage rule constraints (C_{CR}) depend on the predicates in joins, WHERE and HAVING clauses and the grouping attributes in GROUP BY clause. Other elements of SQL that have influence on C_{CR} are relation aliases, aggregation functions and parameters.

Conversion of Predicates into Constraints . Due to the three-valued logic of the predicates in coverage rules, domains of both RO and SD variables include the *null* value and constraints must consider this value and evaluate properly. Consider a simple coverage rule `SELECT * FROM R WHERE a=b` and assume that the values of a and b are NULL. Constraint `a=b` would be satisfied in two-valued logic (a has the same value as b). However, the coverage rule does not return rows because the predicate `a=b` evaluates to unknown or NULL in three-valued logic. Therefore, it is necessary to complement the predicates so that the constraints take into account *null* values as indicated in Table 7. In the example, the constraint must restrict values of a and b to be not NULL; the predicate `a=b` is converted into the constraint $\neg isnull(a) \wedge \neg isnull(b) \wedge a = b$.

Table 7 displays the conversions of coverage rule predicates into constraints. The general rule is to ensure the value of variables is not NULL (using the predicate $\neg isnull(A)$) before evaluating predicates.

Below, the constraints arising from different SQL clauses in coverage rules are defined. In each definition, predicates are considered to establish the constraints according to the previous conversions.

Definition 9.- (*WHERE clause constraint*). Given a coverage rule CR with a select operator $R\,[p(A)]$, the predicate $p(A)$ in the WHERE clause is converted

Table 7 Conversions of predicates into constraints for evaluating three-valued logic

Predicate: p	**Conversion of p into constraints: CONV(p)**
$p(A)$	$\neg isnull(A) \wedge p(A)$
$p(A, B)$	$\neg isnull(A) \wedge \neg isnull(A) \wedge p(A, B)$
$not(p(A))$	$\neg isnull(A) \wedge \neg p(A)$
$p(A)$ and $q(B)$	$\neg isnull(A) \wedge p(A) \wedge \neg isnull(B) \wedge q(B)$
$p(A)$ or $q(B)$	$(\neg isnull(A) \wedge p(A)) \vee (\neg isnull(B) \wedge q(B))$
A IS NULL	$isnull(A)$
A IS NOT NULL	$\neg isnull(A)$
$not(A$ IS NULL$)$	$\neg isnull(A)$
$not(A$ IS NOT NULL$)$	$isnull(A)$

into a constraint that restricts the values a_{RO} of RO variables for the attributes A, as indicated in Table 7:

$$C_{CR}(R[p(A)]) := CONV(p(a_{RO}))$$

Note that the name of the value of attributes is a_{RO} instead of a because these constraints restrict the values of the output tuples of the coverage rule, represented in RO variables.

JOIN Operator Constraints . Consider the coverage rule `SELECT * FROM R INNER JOIN S ON R.ID = S.c`, which is covered when there exist tuples in R and S relations which verify the predicate `R.ID=S.c`. Due to the three-valued logic, as WHERE predicates, it is converted into a constraint following the conversions in Table 7.

In the case where coverage rules have an outer operator, LEFT OUTER JOIN (LJ) or RIGHT OUTER JOIN (RJ), they are covered when at least one row is returned by the inner join or by the outer increment. For example, executing the coverage rule `SELECT * FROM R LEFT JOIN S ON R.ID=S.c` obtains output with tuples in R and S that verify the predicate `R.ID=S.c` or with tuples in R that do not join to any tuple in S and *null* values for the attributes in S. Constraints must ensure one of the following conditions:

1. tuples in RO variables of R and S such that the join predicate is verified,
2. tuples in R and S such that the join predicate is false for values of $R.ID$ in RO variables and any value of $S.c$ in SD variables, and every attribute in S in RO variables is NULL.

When the join type is RJ, constraints are symmetrically established.

Definition 10.- (*INNER JOIN operator constraint*). Given a coverage rule CR with an inner join operator $R[p(A, B)]^{I}S$ and relations R and S in RO variables. Let a_{RO} and b_{RO} be values of attributes of R and S respectively, and $CONV$ a conversion function defined in Table 7:

$$C_{CR}(R[p(A, B)]S) := CONV(p(a_{RO}, b_{RO}))$$

Definition 11.- (*OUTER JOIN operator constraint*). Given a coverage rule CR with an outer join operator $R[p(A,B)]^{JT}S$, where the join type (JT) is LEFT OUTER JOIN (LJ) or RIGHT OUTER JOIN (RJ), and relations R and S in RO and SD variables. Let a_{RO} and b_{RO} be values of attributes of R and S in RO respectively, i and j be tuples of R and S in SD, a^i_{SD} be values of attributes of R in the tuple i, b^j_{SD} be values of attributes of S in the tuples j, and $CONV$ a conversion function defined in Table 7:
If join type (JT) is LJ, for each tuple j of S in SD variables:

$$\begin{aligned} C_{CR}(R[p(A,B)]^{LJ}S) :=& CONV(p(a_{RO}, b_{RO})) \\ &\vee (isnull(b_{RO}) \wedge CONV(not(p(a_{RO}, b^j_{SD})))) \end{aligned}$$

If join type (JT) is RJ, for each tuple i of R in SD variables:

$$\begin{aligned} C_{CR}(R[p(A,B)]^{RJ}S) :=& CONV(p(a_{RO}, b_{RO})) \\ &\vee (isnull(a_{RO}) \wedge CONV(not(p(a^i_{SD}, b_{RO})))) \end{aligned}$$

Framing Constraints . When a query has a GROUP BY clause, HAVING clauses or aggregation functions (avg, sum, max, min, count), applying SQLFpc criterion, several coverage rules are generated to exercise test requirements. Some of them are dependent on the groups and the number of rows in each group that compose the output, and others concerning aggregation functions and the attributes on them. Consider the query `SELECT a,c, sum(b) FROM R WHERE a>1 GROUP BY a,c`; taking into account the different patterns of rules than can be generated, the following cases can be distinguished:

– Case 1.- Coverage rules exercise test requirements related to conditions in the WHERE clause. The rule will be covered if the output is a group with a tuple verifying all predicates within it; in this case, coverage rule constraints, C_{CR}, model the predicates of the rule without aggregate functions although they exist. For the previous query, a coverage rule is `SELECT a,c, sum(b) FROM R WHERE (a=1) GROUP BY a,c`.
– Case 2.- Coverage rules exercise test requirements related to the grouping attributes. There are two sub-cases:
  – Case 2.1: The output must have at least one group formed by at least two rows with the same values in grouping attributes. Considering the query, the coverage rule is `SELECT a,c,sum(b) FROM R WHERE a>1 GROUP BY a,c HAVING count(*)>1`. In this case, constraints must restrict the values such that R in RO variables have different tuples with equal values of grouping attributes a and c.
  – Case 2.2: The output must have at least one group with rows with different values in specific attributes different from the grouping attributes. For the query, a coverage rule is `SELECT c FROM R WHERE a>1 GROUP BY c HAVING count(distinct a)>1`. Constraints establish that R in

RO variables has different tuples with the same value in grouping attribute c but different values in the attribute a.

- Case 3.- Coverage rules exercise test requirements related to attributes in aggregate functions. There are two sub-cases:
  - Case 3.1: The output must have at least one group with tuples that have duplicate and non-duplicate values in the attribute which is in the aggregate function. The coverage rule is `SELECT a, c, sum(b) FROM R WHERE a > 1 GROUP BY a, c HAVING count(b)>count(distinct b) AND count(distinct b)>1`. Constraints restrict the values of R in *RO* variables in order to have different tuples with duplicate and non-duplicate values in the attribute b, which is in the aggregation function `sum(b)`.
  - Case 3.2: The coverage rule for this case is only generated when the attribute in the aggregation function is nullable. The output must have at least one group with tuples where different values in the attribute exist and some of them are equal to NULL. The rule is `SELECT a, c, sum(b) FROM R WHERE a >1 GROUP BY a,c HAVING count(*)>COUNT(b) AND COUNT(DISTINCT b)>1`. Constraints must restrict the values of R in *RO* variables in order to have tuples with non-duplicate and NULL values in the attribute b.

When a query has predicates in the HAVING clause, these predicates will be added to the HAVING clause in each coverage rule generated and they must be satisfied within rows in each group. Suppose the query `SELECT a,c, sum(b) FROM R WHERE a>1 GROUP BY a,c HAVING sum(b)>10 and c>0` and the coverage rule `SELECT a,c, sum(b) FROM R WHERE a>1 GROUP BY a, c HAVING sum(b)>10 and c>0 and count(*)>1`. The output of this coverage rule must have a group formed with at least two rows where `sum(b)` is greater than 10 and values of the attribute c are greater than 0. Therefore, constraints established must ensure these predicates evaluate to true into the groups formed after framing, but bearing in mind that:

- Case 4.- If the predicate has an aggregation function, like `sum(b)`, it is evaluated using the result of executing the aggregation function on the rows.
- Case 5.- Otherwise, in case `c>10`, the predicate is evaluated on each row that forms the group.

Definition 12.- (*Framing constraints*). Given a coverage rule CR with a select operator after framing $R(A)///G[pAF]$, where $G \subseteq A$ is the set of grouping attributes and pAF (predicate After Framing) is the predicate applied after framing which may contain aggregation functions, and a relation R in RO variables. Let $G_i \in G$ be attributes of R, j, k and l be tuples in R, X be an attribute of R, g_i^j be the value of G_i in tuple j, x^j be the value of X in tuple j, and $CONV$ a conversion function defined in Table 7 :
If pAF is `count(*)>1` (case 2.1):

$$C_{CR}(R(A)///G[count(*) > 1]) := \neg(pk(r^j) = pk(r^k)) \wedge g_i^j = g_i^k$$

If pAF is `count(distinct X)>1` (case 2.2):

$$C_{CR}(R(A)///G[count(distinctX) > 1)]) := \neg(x^j = x^k) \wedge g_i^j = g_i^k$$

If pAF is `count(X)>count(distinct X) and count(distinct X)>1` (case 3.1):

$$\begin{aligned} &C_{CR}(R(A)///G[count(X) > count(distinctX) \\ &\qquad \text{and} \quad count(distinctX) > 1]) := \\ &\quad \neg(pk(r^j) = pk(r^k)) \wedge x^j = x^k \wedge \neg(x^j = x^l) \end{aligned}$$

If pAF is `count(*)>count(X) and count(distinct X)>1` and $nl(X)$ (case 3.2):

$$\begin{aligned} &C_{CR}(R(A)///G[count(*) > count(X) \quad \text{and} \quad count(distinctX) > 1]) := \\ &\qquad \neg(x^j = x^k) \wedge isnull(x^l) \end{aligned}$$

If pAF is a predicate in the form $p(aggf(X))$, which does not follow the previous patterns (case 4), where $aggf$ is an aggregation function:

$$C_{CR}(R(A)///G[p(aggf(X))]) := CONV(p(aggf(x^1, x^2 \ldots)))$$

If pAF is a predicate in the form `q(G)` without aggregation functions (cases 1 and 5):

$$C_{CR}(R(A)///G[q(G)]) := CONV(q(g^j))$$

Other features in Coverage Rules . When a coverage rule has an alias defined for a relation in the FROM clause, RO variables are included for tuples of this alias as if it was a relation. Furthermore, an additional constraint is established if there are two or more aliases for the same relation. In this case, there are RO variables to represent tuples for each alias and constraints must restrict the values of these tuples such that if the primary keys are equal, the rest of the values in the attributes must be equal.

Definition 13.- (*Relation alias constraint*). Given a coverage rule CR with $alias_j$ and $alias_k$ relation aliases referencing the same relation R in RO variables. Let A be attributes of R, j and k be tuples corresponding to $alias_j$ and $alias_k$, respectively, and a^i be the value of A in tuple i:

$$C_{CR}(R, alias_j, alias_k) := pk(R_j) = pk(R_k)) \Rightarrow$$
$$(isnull(a^j) \wedge isnull(a^k)) \vee a^j = a^k$$

Another feature of the coverage rule that must be taken into account is that of having parameters. If a coverage rule has them, they are represented with RO variables and they are going to be treated by constraints dependent on the coverage rule, C_{CR}, in the same way as attributes of relations

All constraints presented in previous sections, depending on SQL clauses, are individually established for each clause. However, a coverage rule may be composed of multiple predicates in different clauses. In this case, the coverage rule constraint will be established as the conjunction of all constraints.

Consider the coverage rule `SELECT a,c,sum(b) FROM R WHERE a>1 GROUP BY a,c HAVING sum(b)>10 and c>0 and count(*)>1`. The coverage rule constraint is:

$$C_{CR}(R[a > 1]///G(a, c)[count(*) > 1 \quad \text{and} \quad sum(b) > 10 \quad \text{and} \quad c > 0]) :=$$
$$C_{CR}(R[a > 1]) \wedge C_{CR}(R///G(a, c)[count(*) > 1])\wedge$$
$$C_{CR}(R///G(a, b)[sum(b) > 10]) \wedge C_{CR}(R///G(a, b)[c > 0])$$

Definition 14.- (*Composition constraint*). Given a coverage rule CR and C^i_{CR} constraints for predicates in different clauses of CR:

$$C_{CR} := C^1_{CR} \wedge C^2_{CR} \dots$$

4.2.4 State Transformation Constraints, C_{ST}

Constraints previously defined, C_{DS}, C_S and C_{CR}, are specifically focused on the database, the schema or coverage rules. However, maintaining the relation between SD and RO variables is necessary due to the fact that the output tuples of a coverage rule (represented by RO variables) depend on the information stored in the database (represented by SD variables). State Transformation Constraints (C_{ST}) relate both types of variables, which allow the database state DS_{i-1} to be transformed into a new database state DS_i.

Constraints C_{ST} ensure tuples in RO variables exist in SD variables, meaning that tuples in the output of the coverage rule are in the database. These constraints take into account that the primary key of each tuple of a relation in RO implies the existence of the same tuple of the relation in SD and if a tuple in RO variables does not exist in SD variables, it will be created.

Definition 15.- (*State transformation constraint*). Given a relation R in SD and RO variables. Let A_{SD} and A_{RO} be attributes of a relation R in SD and RO, j and k be tuples of R in SD and RO, and a^j_{SD} and a^k_{RO} be values of attributes A_{SD} and A_{RO} in tuples j and k:

$$C_{ST}(R) := pk(R^j_{SD}) = pk(R^k_{RO}) \Rightarrow (isnull(a^j_{SD}) \wedge isnull(a^k_{RO})) \vee a^j_{SD} = a^k_{RO}$$

Consider, for example, the coverage rule `SELECT * FROM R WHERE R.b IS NULL` and the database state DS_{i-1} where $R(ID, a, b) = \{(1, 5, \oslash)\}$. Suppose, initially, variables SD and RO are instantiated as $SD = \{(1, 5, \oslash)\}$ and $RO = \{(\oslash, \oslash, \oslash)\}$. The coverage rule constraint, C_{CR}, related to the WHERE condition `R.b IS NULL`, is satisfied by instantiating RO variables to $\{(\oslash, \oslash, null)\}$. State transformation constraints, C_{ST}, could be satisfied (tuples of the output of the coverage rule are in the database) in two different ways:

1. Adding a new tuple to SD variables with $null$ value in the attribute $R.b$ and using its $R.ID$ value in RO variables. In this case, the solution is $SD = \{(1, 5, \oslash), (2, \oslash, null)\}$ and $RO = \{(2, \oslash, null)\}$.
2. Reusing a tuple of SD variables: completing the tuple in RO variables with $R.ID = 1$ and $R.a = 5$ (values in the existing tuple of SD variables), and assigning $null$ value to the attribute $R.b$ in the tuple in SD. The solution is $DS = \{(1, 5, null)\}$ and $RO = \{(1, 5, null)\}$.

4.3 Strategy for Finding a Solution

Given a coverage rule CR and an initial database state DS_{i-1}, the database state generation problem is stated as CSP. If it is possible to find the solution verifying all the constraints, the solution is a new database state DS_i, which is the result of the transformation function $T(DS_{i-1}, CR)$. When the coverage rule has parameters, the solution includes the database state DS_i and the set of pairs $\langle param, value\rangle$, where $param$ is the name of parameters in the coverage rule and $value$ is its corresponding value assigned in the solution, in order to evaluate to true predicates of the coverage rule. Otherwise, if the constraints cannot be satisfied, no solution is found for the problem and the coverage rule is not covered starting from DS_{i-1}.

Our purpose is focused on obtaining meaningful databases that are as reduced in size as possible. Reusing information and instantiating attributes without value in the database state DS_{i-1} is the key to obtaining databases which avoid inserting unnecessary new tuples. For this reason, we are going to use an optimization strategy.

As an illustrative example, consider the coverage rule `SELECT * FROM R INNER JOIN S ON R.ID=c` and the database state DS_{i-1} where $R(ID, a, b) = \{(1, \oslash, 11)\}$ and $S(ID, c, d) = \{(1, \oslash, \oslash)\}$. There are different alternatives to transform the database state DS_{i-1} into DS_i in order to cover the coverage rule, but depending on the generation strategy, the solution changes:

– Alternative 1: the strategy is to always insert new tuples. In this case DS_i is: $R = \{(1, \oslash, 11), (2, \oslash, \oslash)\}, S = \{(1, \oslash, \oslash), (2, 2, \oslash)\}$.

– Alternative 2: the strategy is to reuse existing tuples (without modifying them) and insert new tuples in other cases. DS_i is: $R = \{(1, \oslash, 11)\}, S = \{(1, \oslash, \oslash), (2, 1, \oslash)\}$.
– Alternative 3: the strategy is to reuse tuples, instantiate attributes and insert new tuples in other cases. DS_i is $R = \{(1, \oslash, 11)\}, S = \{(1, 1, \oslash)\}$.

We use an optimization strategy based on the minimization of the number of tuples of the relations in SD variables. To this end, we have defined a minimization function for the sum of the number of tuples of the relations in SD variables that it should be applied in order to solve the problem in an optimized way.

Definition 16.- (*Minimization function for database state generation*). Given the relations R_i in SD variables. Let $\#R_i$ be the number of tuples of R_i, the minimization function is defined as $\min(\sum(\#R_i))$.

Continuing with the above example, the minimization function will evaluate the lowest value with the third alternative where the database state has two tuples (whereas database states in alternatives 1 and 2 have four and three tuples, respectively).

5 Tool Support: QAGrow

We have developed the QAGrow Tool (Query Aware Grow databases) that fully automates our approach by implementing the algorithm described in Sect. 3.3. It includes the search for the solution to the test database generation problem (**Definition 1**).

For finding the solution to the database state generation problem for a coverage rule (**Definition 2**), we have integrated *Choco* (Prud'homme et al 2015), version 2.1.2, into our tool. It is a free and Open-Source java library, whose implementation embeds and internally manages a SAT (Boolean **SAT**atisfiability Problem) solver. It builds on an event-based propagation mechanism with backtrackable structures.

In QAGrow tool, using *Choco*, the database state generation problem for a coverage rule is modeled in the form of Constraint Satisfaction Problem and it is solved with Constraint Programming Techniques. *Choco* launches a resolution, uses its default search strategy and stops at the first solution found. The optimization strategy is established with the aim of minimizing the number of tuples generated in the test database.

5.1 Tool Description

Fig. 3 depicts the whole process which has an initial database state and a set of queries as inputs, and the set of database instances and the sets of pairs $\langle param, value\rangle$ that cover the input queries as outputs. For each query in the input, QAGrow obtains the set of coverage rules using the SQLFpc web service (Tuya et al 2010). For each rule, a pre-processing is carried out, trying to find a solution for the rule from an empty database state. If a solution is found, the rule is feasible and the process continues. However, if no solution is found, the rule is marked as unsatisfiable and its processing ends. After this, it searches for the solution to the database state generation problem (**Definition 2**) considering a database state DS_{i-1}:

1. Modeling the problem as a CSP: SD and RO variables are defined in their domains and constraints are established.
2. Solving the problem using the *Choco* solver.
3. Generating a new database state DS_i and a new set of pairs $\langle param, value\rangle$ if a solution is found. Otherwise, marking the rule as not covered.

After all rules have been processed, QAGrow generates a database instance that includes the last database state with all values generated. If there are still uncovered rules (a subset of CRs), they are processed again, repeating the procedure but starting with an empty database state and obtaining as output a new database instance and new sets of pairs $\langle param, value\rangle$.

5.2 Tool Limitations

QAGrow tool automates our approach for relational databases and SQL statements that read the information stored. It is able to handle a large set of SELECT syntax, including the main clauses (SELECT, JOIN, WHERE, GROUP BY, HAVING) as well as parameters, arithmetic expressions, aggregation functions (avg, sum, max, min, count) and views. In this version, subqueries are not supported. The CASE operator is supported when it is placed in the SELECT clause due to the coverage rules related to present CASE conditions in the WHERE clause as if they were normal WHERE conditions.

Other SQL statements (INSERT, UPDATE, DELETE) that update databases are not directly supported. However, their treatment could be feasible if they were transformed into SELECT queries and were processed similarly to Zhou and Frankl (2011) approach: deriving queries from updating statements then characterizing the state change that would occur if they were executed.

Regarding database schema constraints, QAGrow tool supports primary keys, foreign keys and nullability checks. Other features of Database Management Systems (DBMS), such as stored procedures, triggers or other types of check restrictions, which could include pieces of code different from SQL queries (for example, PL/SQL in Oracle) or present a variety of structures depending on the DBMS, are not yet handled.

As QAGrow tool uses the constraint solver *Choco*, the features of *Choco* limit some aspects of the tool. In *Choco*, a variable domain can be integer, boolean, set and real, but it does not include support for strings. Our implementation generates integers instead of strings when the database schema includes the type of attributes as strings, therefore the LIKE operator is not handled by this tool.

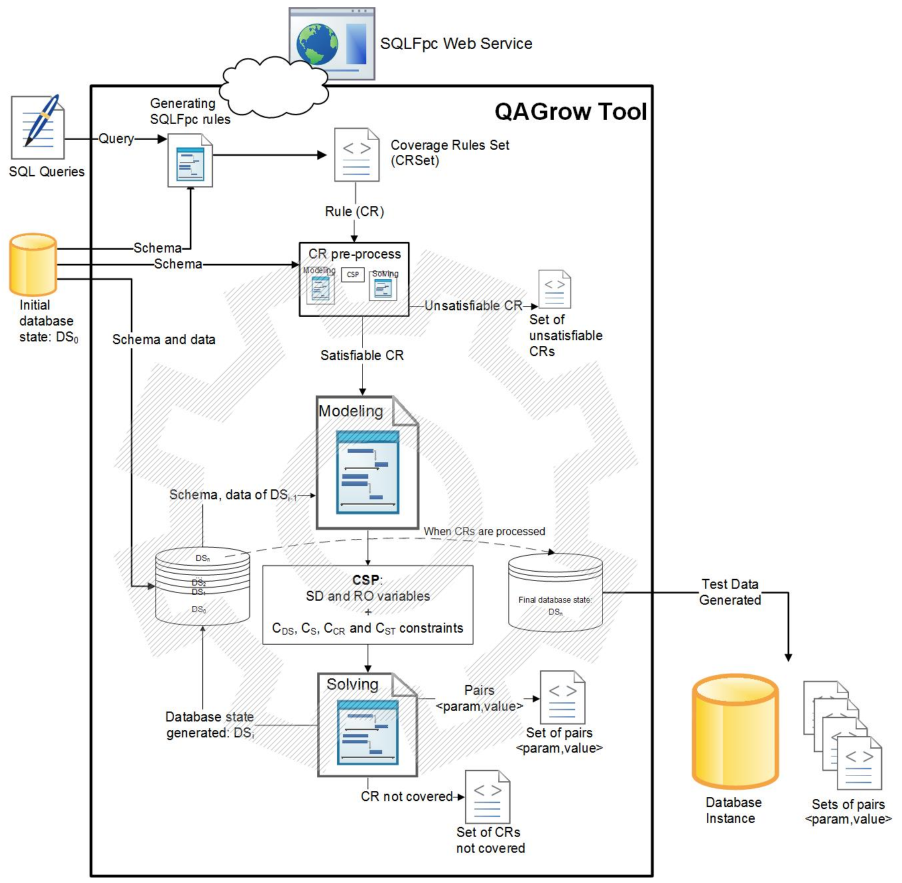

Fig. 3 QAGrow Tool

This work is focused on the coverage at the query level. Therefore, the program under test is a set of SQL queries and test requirements that are obtained by applying the SQLFpc coverage criterion (Sect. 2.2) to the queries. This is well suited for reporting when most of the application logic resides in queries, but in general does not guarantee the coverage of the procedural code of an application. In programs that use databases, part of the application logic is embedded in the SQL queries that access the database and some decisions taken in the code depend on the result of queries executed. The more application logic is embedded in queries, the more likely it is that coverage of the procedural code be kept because the generated database has considered the coverage of the logic embedded in queries. Experiments in (Tuya et al 2016) considered a program under test that includes four methods coded in Java with 15 decisions and 9 queries dynamically constructed. The decisions were based on the output of these queries. They showed the execution of the designed test cases, including reduced databases that fulfilled SQLFpc criterion, led to 100% of decision coverage.

An alternative usage scenario is to generate test databases to check how the application behaves when queries do not get information from databases. In this case, there are two ways to carry this out using QAGrow tool: (1) before the generation process, select coverage rules whose output is different from the original query output and generate test databases for them; or (2) after the generation process, filter information in test databases that would not be selected by queries, and then execute queries against the filtered information. In both cases, no row will be returned when queries of applications are executed.

6 Experiments and Results

In order to evaluate the feasibility of our approach for test database generation, our evaluation addresses the following research questions:

RQ1: How many database instances and how many rows are generated?

RQ2: What is the performance of the approach comparing the databases generated with other existing test databases in terms of effectiveness, coverage and fault-detection?

RQ3: What is the cost of the process taking into account the time spent?

6.1 Experiment Setup

An application named Helpdesk, a real-life web system that manages user support requests (known as *tickets*), has been used for evaluating our approach. The database contains 37 tables and the main information stored is the helpdesk ticket, which is created for each user request. Whenever an action is performed on a ticket, a history record is created. The application implements a complete security subsystem that, before starting each transaction, executes a set of the SQL queries embedded in the procedural code to check permissions. We have taken the set of queries from the database logs collected during exploratory testing sessions to check the security subsystem.

From those queries, and according to complexity in the number of tables and joins and in the number of conditions, we have considered two different subsets of queries for the evaluation of the generation process of test databases:

– Q_s is a set of 20 simple queries selected, where the maximum number of tables in joins is 2 and the maximum number of conditions is 4.
– Q_c is a set of 195 complex queries selected which have joins up to 10 tables, up to 19 conditions and query parameters.

Given these sets of queries, we have performed several test database generation processes using the QAGrow tool that implements the approach described in this paper. The test database instances generated allow covering all coverage rules. The generation processes and resulting test databases are:

– Starting from an empty database state, using the queries in Q_s, we obtain a set of two test database instances, DB_s.
– Starting from an empty database state, using the queries in Q_c, we obtain a set of three database instances DB_c and sets of pairs $\langle param, value\rangle$ for rules that have parameters to instantiate them during the execution of the tests.
– Starting from the first database instance of the set DB_s generated previously, DB_{s1}, using the queries in Q_c, we obtain a set of three database instances DB_c^{s1} and sets of pairs $\langle param, value\rangle$.

Experiments have been run on an Intel®Core ™i7, 2.30GHz. with 8 GB of memory using the QAGrow tool for generating the test databases, and the SQLFpc v. 1.1.84.0 and SQLMutation v. 1.2.77.2 web services for evaluating the coverage and the mutation score of the queries under test against the test databases.

6.2 Analysis and Comparison of the Results

In order to analyze and compare the fault detection ability and the coverage of generated test databases, we have taken:

– A copy of the production database, named *prodDB*, that is used to measure the coverage and the mutation score for both sets of queries. It was used during exploratory testing sessions from which queries for evaluation were taken from the database logs.
– A database generated using a global approach described in our previous work (de la Riva et al 2010) using the constraint solver *Alloy*[1], named *alloyDB*. It can only be used to measure the coverage and the mutation score for Q_s because that approach does not support queries as complex as those of Q_c.

Next, the results obtained in the generation processes are analyzed and compared with each of the aforementioned databases. Discussions are focused on answering the research questions, which relate to the number of generated database instances, the effectiveness of the approach (percentage of coverage and mutation score) and the generation time.

6.2.1 Test Databases Generated

Table 8 summarizes the results of test databases generated including the number of rows of the main tables (tickets, history records and users), the total number of rows (#rows), the time (in seconds) of each generation and the number of coverage rules (#rules) that have been covered in the generation of test databases. As results produced are sets of database instances, there is one column for each set of test databases generated (DB_s, DB_c and DB_c^{s1}) that

[1] http://alloy.mit.edu/alloy/

Table 8 Test database generation results for the sets of queries Q_s and Q_c

	Test DBs Generated by QAGrow			**Test DBs for comparing**	
	DB_s	DB_c	DB_c^{s1}	*prodDB*	*alloyDB*
#tickets	12	50	59	22,387	8
#hrecords	4	2	4	103,553	4
#users	10	19	21	279	7
#rows	**82**	**200**	**238**	**139,259**	**139**
Time(s)	**1.66**	**118.72**	**116.08**	**-**	**285**
#rules	**68/68**	**1,269/1,271**	**1,269/1,271**	**42/68; 717/1,271**	**58/68**

Table 9 Test database instances generated for the sets of queries Q_s and Q_c

	Test DBs Instances Generated		
	DB_s	DB_c	DB_c^{s1}
#tickets	10+2	50+0+0	39+20+0
#hrecords	3+1	2+0+0	4+0+0
#users	9+1	19+0+0	9+12+0
#rows	69+13	188+10+2	176+59+3
Time(s)	1.50+0.15	118.66+0.05+0.01	106.31+9.75+0.02
#rules	54+14	1,260+8+1	1,081+186+2

contains the sum of: (1) rows, (2) the generation time of the instances and (3) the covered and total coverage rules (covered/total). Last two columns include the values in databases used for comparing (*prodDB* and *alloyDB*). Please note that the #rules cell for *prodDB* has two values: covered and total rules using queries in Q_s (42/68) and using queries in Q_c (717/1,271). Moreover, Table 9 contains one column for each set that presents these values disaggregated for each instance (values are separated by '+').

When test databases are generated starting from an empty database state (DB_s and DB_c), the number of rows for both sets of queries, taking into account the rows of all instances (82 rows in DB_s and 200 rows in DB_c), is significantly less than *prodDB* (139,259 rows). Therefore, generated test databases may contribute to avoiding problems associated with handling large amounts of data (DB_c is only 0.14% of *prodDB*).

For the set of queries Q_s, comparing DB_s (82 rows) with the test database obtained with a global approach (*alloyDB* with 139 rows), we can highlight that QAGrow: (1) optimizes the generation with a lower number of rows (DB_s is 58.99% of *AlloyDB*) and (2) is able to generate other test database instances to cover rules not yet covered. The *Alloy* approach is not able to generate more than one test database, and 10 coverage rules remain uncovered with *alloyDB*.

Most of the coverage rules are covered with a single database instance. However, it is necessary to generate others, in general smaller than the first one, in order to cover a few test requirements that are inconsistent with the others in the same database instance. After generating the first database instance for Q_s, 14 coverage rules of 68 are still pending for covering which will be covered with a new database instance. In the case of the set Q_c starting from an empty database state, after generating the first database instance, 11 rules of 1,271 are still uncovered; the second database instance is generated to cover 8 rules and the third instance for only one rule. There are two coverage rules which are not covered with any instance; these are unsatisfiable rules where conditions in the WHERE clause are impossible to evaluate true. In this clause, there are unsolvable expressions due to repeated coupled conditions, in the form `NOT(A=1 OR A=2) AND NOT(A<>1 AND A<>2)`. These types of rules are detected automatically by QAGrow, which carries out a pre-processing in order to ensure rules can be covered in the generation process.

When test databases are generated starting from a non-empty state, results (DB_c^{s1}) are finally similar to the above point (DB_c) although they are not the same during the process: in both cases the number of rows generated is less than in *prodDB*, however, the distribution of rows in the database instances are quite different. The first database instance generated in the set DB_c^{s1} has slightly fewer rows (176 rows) than that generated beginning with an empty database state (188 rows). This is due to the fact that rows in the non-empty initial database state add constraints which make it impossible to cover more coverage rules. Therefore, the second database instance generated contains more rows (59 rows) in order to cover these coverage rules.

In conclusion, answering RQ1, QAGrow generates a database instance with a low number of rows, which cover as many coverages rules as possible, and when there are incompatible rules, it generates additional database instances that allow covering all rules.

6.2.2 Effectiveness

The analysis of the results considers two dependent variables: (1) the percentage of SQLFpc coverage (Tuya et al 2010) reached for each set of queries and databases and (2) the mutation score calculated using SQLMutation (Tuya et al 2007) to compare the effectiveness in detecting faults of each database.

Table 10 contains information about coverage and mutation test scores: the number of coverage rules (#CovR) and mutants (#Mut) generated from the sets of queries Q_s and Q_c, and the percentage of coverage rules covered (%SQLFpc) and the mutation score (%MutScore) reached using different test databases (generated by QAGrow, DB_s, DB_c, and DB_c^{s1}, and used for comparing, *prodDB* and *alloyDB*). Due to the fact that QAGrow generates sets of database instances, %SQLFpc and %MutScore are calculated by accumulating the results obtained from the execution against each instance. In Table 11, columns "Disaggregated %SQLFpc" and "Disaggregated %MutScore" show the scores disaggregated in each test database instance. Note that the sum of values in these columns is not equal to values in columns "%SQLFpc" or "%MutScore" because a coverage rule or a mutant could be covered by more than one instance. For example, for the set Q_s, there are 11.77% of coverage rules that are covered by both database instances of DB_s.

Table 10 SQLFpc Coverage and Mutation Score

Queries	Test Databases	SQLFpc		SQLMutation	
		#CovR	**%SQLFpc**	**#Mut**	**%MutScore**
Q_s (20 simple queries)	DB_s (2 instances, 82 rows)	68	**100**	2,417	**70.09**
	prodDB (1 inst., 139,259 rows)		**61.76**		**65.54**
	alloyDB (1 instance, 139 rows)		**85.33**		**84.13**
Q_c (195 complex queries)	DB_c (3 inst., 200 rows)	1,271	**98.98**	56,787	**80.44**
	DB_c^{s1} (3 inst., 238 rows)		**98.98**		**80.12**
	prodDB (1 inst., 139,259 rows)		**56.41**		**60.06**

Table 11 SQLFpc Coverage and Mutation Score of test database instances generated

Queries	Test Database	Disaggregated %SQLFpc	Disaggregated %MutScore
Q_s	DB_s (2 instances, 82 rows)	91.18;20.59	69.22; 11.63
Q_c	DB_c (3 instances, 200 rows)	98.27; 12.27; 7.32	80.30;0.71;0.26
	DB_c^{s1} (3 instances, 238 rows)	86.78;35.09;6.61	71.45; 35.24; 0.28

Regarding the SQLFpc coverage criterion, test databases generated by QAGrow reach the highest possible coverage scores (100% for Q_s, 98.98% for Q_c). These results are larger than those obtained with the production database (61.76% for Q_s, 56.41% for Q_c), even though *prodDB* has a much higher number of rows. Compared with results obtained by a global approach (test database *alloyDB*), due to the optimization strategy, QAGrow is able to populate test databases that not only reach higher coverage scores but also have fewer rows than *alloyDB* (100% and 82rows vs. 85.33% and 139rows).

Regarding the mutation analysis, results show %MutScore is higher in databases generated with QAGrow than *prodDB*, therefore the fault detection effectiveness can be improved by using generated test databases instead of production databases (70.09% with DB_s vs. 65.54% with *prodDB* for Q_s, up to 80% with DB_c vs. 60.06% with *prodDB* for Q_c).

However, the mutation score of DB_s is less than that reached with *alloyDB* (70.09% with DB_s vs. 84.13% with *alloyDB*). This is specifically due to NLS and IRC mutants which replace columns in SELECT clauses (score of NLS: 10% with DB_s vs. 54.54% with *alloyDB*; score for IRC: 57.78% with DB_s vs. 85.86% with *alloyDB*). Those types of mutants tend to be killed more easily the greater the diversity of data. The minimization of the number of

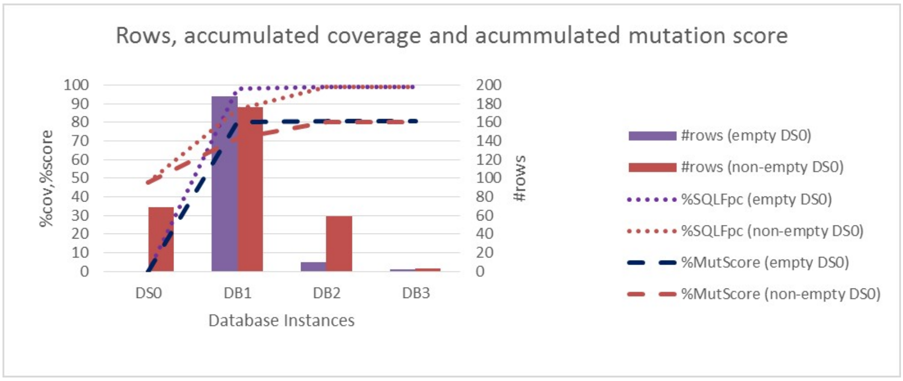

Fig. 4 Number of rows in each generated database instance for the set of queries Q_c and %SQLFpc and %MutScore accumulated with each database, starting from an empty database state and from a database instance (DB_s) generated previously.

rows in databases generated with QAGrow might produce less diverse data and, therefore, less mutants are killed.

Fig. 4 depicts how the coverage and the mutation score evolve as database instances are generated by QAGrow for the set of queries Q_c when the process starts from an empty database state in order to generate test database DB_c and when it starts from a non-empty initial database state (DS0) to generate DB_c^{s1}. For DB_c^{s1}, the initial coverage and the initial mutation score are close to 50% on its DS0 and they are 0% for DB_c. However, after generating the first database instance (DB1), these scores are higher and quite close to the maximum for DB_c that begins with an empty database state (98.27% vs. 86.78% for coverage and 80.30% vs. 71.45% for mutation score). For the rest of the database instances generated, the accumulated coverages and the mutation scores are very similar for both cases. From the figure, we can appreciate that QAGrow allows us to generate test databases obtaining high scores of coverage and a mutation score independent of the initial database state, although better results are reached earlier when the process starts from an empty database state.

Moreover, the maximum coverage reached for the set Q_c is 98.98% is not the maximum possible (99.84%) due to the loss of coverage because there are 11 coverage rules which are not covered at the end of the process. These rules are covered during the generation process. However, rows in master tables without details, which allowed their coverage, are re-used adding details for covering subsequent rules. This situation leads to subsequent rules being covered whilst losing the coverage of previous ones.

In conclusion, answering RQ2, the results of these experiments show the viability of the approach in populating test databases for a set of SQL queries, attaining good scores in the coverage and fault detection ability. This implies that the generated test databases contain good, diverse sets of rows (in the sense that they exercise the target queries) that are good enough to be used for testing purposes (in the sense of fault detection capability).

6.2.3 Cost

In Table 8, the Time row displays the number of seconds spent during the generation processes and, in Table 9, the time spent in obtaining each database instance (separated by '+').

We do not compare the time values with the time it takes *prodDB* because it was obtained from the production environment. However, we can compare the time of generating databases with QAGrow with the global *Alloy* approach.

For the set of queries Q_s, generating the set of database instances starting from an empty database state takes less time than generating the *alloyDB* test database with a global approach (1.66s. vs. 285s.). More importantly, QAGrow populates database instances with fewer rows (82rows vs. 139rows) more meaningfully from the point of view of the coverage (100% vs. 85.33%). Therefore, there is a remarkable improvement over previous work (de la Riva et al 2010).

For the set of queries Q_c, the generation process starting from an empty database state takes approximately the same time as starting from a non-empty state (118.72s. vs. 116.08s.).

Determining which of the two test database generations (starting from an empty database state or beginning with a non-empty state) would be more adequate does not depend on the generation time or on the accumulated coverage (because they are the same). Only if the tester decided to manage a single database instance, starting from an empty database state would be the best option because, although more rows are generated (188rows vs. 176rows), both the coverage and the mutation score are higher (98.27% vs. 86.78% for %SQLFpc and 80.30% vs. 71.45% for %MutScore).

Answering RQ3, the incremental approach generates test database instances taking fewer seconds than the global approach (de la Riva et al 2010) and the test databases are smaller and more significant, achieving higher coverage.

7 Related Work

Database testing is a challenging problem, which has garnered renewed attention in recent years. With regard to functional testing, which is the scope of this paper, most works focus on test input or test database generation over the SQL queries as well as coverage evaluation, the database application or the database management systems. Below we discuss the work on these topics.

One of the first works on testing SQL statements using automated reasoning was the work in Zhang et al (2001). Given an SQL query, the schema and a set of user test requirements as input, the output is a set of constraints that define the characteristics of the test database. Database instances for testing the SQL query can be derived by solving these constraints. Using this idea as a basis, other works explore the same problem. Khalek et al (2008) define a tool for test data generation incorporating *Alloy* specifications both for the schema and for the query. Each table is modeled as a relation over the attribute domains and the query is specified as a set of constraints that models the condition in the WHERE clause. As test criterion, they use a predicate coverage criterion over the predicate of the WHERE clause with the goal of generating a test database and validating the output of queries in different DBSMs. The scalability is limited because the approach cannot handle tables with a larger number of rows (no more than four according to their experiments) and tables have at most two attributes. In contrast, our approach supports larger schemas and databases, and we use a specific test criterion for SQL queries that takes into account the particularities of the SQL language in addition to WHERE predicates (e.g. JOIN operators, nullable values). In Veanes et al (2009), the satisfiability modulo theories solver Z3 has been used to generate input data for SQL queries satisfying a given test condition. Whereas the test conditions are given in an ad-hoc fashion (the query result is empty, nonempty, contains a value, etc.), our approach employs automated and query-based test conditions (SQLFpc coverage rules) to guide the database generation. Binnig et al (2007a) propose a technique named Reverse Query Processing for generating test databases that takes the query and the desired output as input and generates a database instance (using a model checker) that could produce that output for the query. This approach supports one SQL query and therefore generates one test database for each query. A further extension to this work (Binnig et al 2008) supports a set of queries and allows specifying to the user the output constraints in the form of SQL queries. However, the creation of these constraints could be difficult if the source specification is not complete. Caballero et al (2010) developed a framework for generating test cases for correlated SQL views using CLP (Constraint Logic Programming). Compared with ours, the approach does not support JOIN operators and nullable values and it has not been evaluated over large schemas and databases. Other approaches (Shah et al 2011; Pan et al 2013; Vemasani et al 2014; Chandra et al 2015) use mutation analysis over the SQL queries as test criterion in order to guide the test database generation. The aim is to generate sufficient test data that detect faults in SQL mutants (the original query modified with a defect). In our work, we use mutation analysis as an effectiveness measure rather than as test criteria. Kapfhammer et al (2013) define a technique that generates test data with the aim of testing the integrity constraints. Thus, they focus on testing the relational schema instead of the SQL queries, although the approach could be used in a complementary fashion with our method.

In general, our work differs from preceding works in the following points: (1) We propose a fully automated approach for testing SQL (query parameters and test database), (2) we use a specific test coverage criterion for generating the test database that is specially tailored for SQL queries and (3) our approach supports complex queries over large and complex databases.

On a related but complementary level, testing database applications considers the control flow of the program interacting with the database instead of individual SQL queries. The AGENDA toolset (Chays et al 2004, 2008) describes an approach that populates databases with test data that satisfies the schema constraints. It uses the category-partition method over the schema constraints and uses heuristics to fill the test database. It requires tester intervention to provide the data group whereas our work relies on the SQLFpc coverage rules and thus the test input generation process is fully automatic. Other approaches are based on Dynamic Symbolic Execution (DSE) (Sen et al 2005), which extends traditional symbolic execution by running the program with concrete inputs while collecting both specific and symbolic information at runtime.

Willmor and Embury (2006a,b) develop a technique to specify intensional test cases for database applications. The database test cases are formed by preconditions that specify the initial state of the database and post-conditions that must hold after execution of the target program and are provided by the user. Symbolic execution was used to generate input and database state. While the post-conditions are outside the scope of this work, the preconditions have a similar purpose to the SQLFpc coverage rules we use, but in our approach they are automatically generated. Other works (Emmi et al 2007; Pan et al 2014; Marcozzi et al 2015) have been developed to handle the presence of SQL statements within the classical code to be symbolically executed. The main idea is to track symbolic constraints from the procedural code and the embedded SQL queries and then use these constraints in conjunction with a constraint solver to generate program inputs and/or database states. As test criteria, they use branch coverage and the test situations are obtained both from the conditional statements of the procedural code and the conditions of the WHERE clause. Whereas the previous works are addressed to generate test data from scratch, Pan et al (2011, 2015) and Li and Csallner (2010) propose DSE-based approaches for generating program inputs for testing database applications, but they use existing database states in the program input generation in order to avoid the overhead of generating new database states during test generation. In this sense our approach follows the same principle, but we use the database states not only to generate program inputs (in our case query parameters) but also to generate test database states. With a different approach, Blanco et al (2012) develop a specification-based approach to guide the design of the test input and the test database in applications with user-database interactions. Both the application database and the user interface are integrated in a single model and the test requirements are derived from the model in the form of SQLFpc coverage rules. In this context, our approach can be used to automatically generate the test inputs using the test requirements derived from the model as input in our method.

A variety of methods and tools have also been used for testing some features (both functional and extra-functional) and the benchmarking of Database Management Systems (DBMS). Bruno and Chaudhuri (2005) and Houkjær et al (2006) present a data generator that helps the tester to define synthetic databases with rich data distributions with inter- and intra-table relationships. However, in contrast with our work, these methods do not use the information of the queries while generating the data and hence the execution of the query against the generated database might not return results. QAGen (Binnig et al 2007b) and MyBenchmark (Lo et al 2014) are notable examples of query-aware generators. Whereas QAGen supports the data generation for one query, and hence for n queries n independent test databases, MyBenchmark, which uses the QAGen as core, generates a minimal set of test database instances for a set of queries. Other works (Bruno et al 2006; Khalek and Khurshid 2011) support the testing of DBMS by creating SQL queries, instead of generating test data, in order to produce results with different sizes. The main aim of these works is the support of the test database engine (e.g. generation of workload for stress testing and application-based benchmarking) instead of the testing of functional requirements embedded in the SQL queries, which is the primary objective of our method.

8 Conclusions

We have presented an automated approach that takes a set of SQL queries and an initial database state (which can be empty or populated with data) as input and generates a reduced number of test databases with meaningful data and of a reduced size for testing all the queries.

The approach supports a large set of SQL clauses including SELECT, JOIN, GROUP, WHERE and HAVING clauses, as well as the generation of test data for numerical data types.

The results from the experimental evaluation show the feasibility of the approach in generating test relational databases with a high coverage (SQLFpc test coverage) and fault detection ability measured in terms of the mutation score (SQLMutation) for a number of non-trivial SQL queries over a large schema database. The number of generated test databases for a large set of SQL queries is small and with one test database it is possible to achieve high coverage for most of the queries. Additionally, the test databases have a reduced size making the evaluation of the test outputs easier.

Typical scenarios for the application of the approach include either the test database generation starting from scratch or the test generation starting from a previous populated database in order to complete the tests. Because the approach is fully automated, a first evident benefit is the reduction of the time needed to create the test relational database, so the tester is not required during the test data preparation. Moreover, the test data generation is guided by a systematic test criterion ensuring that the test database contains meaningful data to test the queries. In addition, the reduced size of the generated test database contributes to facilitating the task of checking the actual test output when testing the SQL queries, making the process of test output evaluation more reliable.

Future work is addressed to extending the support to other clauses and database schema restrictions (i.e. check constraints) and to considering the test data generation for non-numerical types (i.e. string values) by means of the integration with a string solver. Moreover, QAGrow tool can be adapted to use at application level for checking the application behavior when queries do not return any rows from the database or generating test cases considering test requirements of both procedural code and SQL queries.

Acknowledgements This work was supported in part by projects TIN2013-46928-C3-1-R and TIN2016-76956-C3-1-R, funded by the Spanish Ministry of Science and Technology, and GRUPIN14-007, funded by the Principality of Asturias (Spain) and ERDF funds.

References

Binnig C, Kossmann D, Lo E (2007a) Reverse query processing. In: Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on, pp 506–515

Binnig C, Kossmann D, Lo E, Özsu MT (2007b) Qagen: generating query-aware test databases. In: Chan CY, Ooi BC, Zhou A (eds) Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007, ACM, pp 341–352

Binnig C, Kossmann D, Lo E (2008) Multi-rqp: Generating test databases for the functional testing of oltp applications. In: Proceedings of the 1st International Workshop on Testing Database Systems, ACM, New York, NY, USA, DBTest '08, pp 5:1–5:6

Blanco R, Tuya J, Seco R (2012) Test adequacy evaluation for the user-database interaction: A specification-based approach. In: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, pp 71–80

Bruno N, Chaudhuri S (2005) Flexible database generators. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB Endowment, VLDB '05, pp 1097–1107

Bruno N, Chaudhuri S, Thomas D (2006) Generating queries with cardinality constraints for dbms testing. IEEE Trans on Knowl and Data Eng 18(12):1721–1725

Caballero R, García-Ruiz Y, Sáenz-Pérez F (2010) Applying constraint logic programming to sql test case generation. In: Proceedings of the 10th International Conference on Functional and Logic Programming, Springer-Verlag, Berlin, Heidelberg, FLOPS'10, pp 191–206

Chandra B, Chawda B, Kar B, Reddy K, Shah S, Sudarshan S (2015) Data generation for testing and grading sql queries. The VLDB Journal 24(6):731–755

Chays D, Deng Y, Frankl PG, Dan S, Vokolos FI, Weyuker EJ (2004) An agenda for testing relational database applications: Research articles. Softw Test Verif Reliab 14(1):17–44

Chays D, Shahid J, Frankl PG (2008) Query-based test generation for database applications. In: Proceedings of the 1st International Workshop on Testing Database Systems, ACM, New York, NY, USA, DBTest '08, pp 6:1–6:6

Chilenski JJ (2001) An investigation of three forms of the modified condition decision coverage (mcdc) criterion. Tech. rep., Office of Aviation Research

Codd EF (1990) The Relational Model for Database Management: Version 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Emmi M, Majumdar R, Sen K (2007) Dynamic test input generation for database applications. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA '07, pp 151–162

Halfond W, Orso A (2006) Command-form coverage for testing database applications. In: Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on, pp 69–80

Houkjær K, Torp K, Wind R (2006) Simple and realistic data generation. In: Proceedings of the 32Nd International Conference on Very Large Data Bases, VLDB Endowment, VLDB '06, pp 1243–1246

Kapfhammer G, McMinn P, Wright C (2013) Search-based testing of relational schema integrity constraints across multiple database management systems. In: Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, pp 31–40

Kapfhammer GM, Soffa ML (2003) 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, ACM, New York, NY, USA, ESEC/FSE-11, pp 98–107

Khalek S, Khurshid S (2011) Systematic testing of database engines using a relational constraint solver. In: Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on, pp 50–59

Khalek SA, Elkarablieh B, Laleye YO, Khurshid S (2008) Query-aware test generation using a relational constraint solver. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy, IEEE Computer Society, pp 238–247

Li C, Csallner C (2010) Dynamic symbolic database application testing. In: Proceedings of the Third International Workshop on Testing Database Systems, ACM, New York, NY, USA, DBTest '10, pp 7:1–7:6

Lo E, Cheng N, Lin WW, Hon WK, Choi B (2014) Mybenchmark: Generating databases for query workloads. The VLDB Journal 23(6):895–913

Marcozzi M, Vanhoof W, Hainaut JL (2015) Relational symbolic execution of {SQL} code for unit testing of database programs. Science of Computer Programming 105:44 – 72

Pan K, Wu X, Xie T (2011) Generating program inputs for database application testing. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, Washington, DC, USA, ASE '11, pp 73–82

Pan K, Wu X, Xie T (2013) Automatic test generation for mutation testing on database applications. In: Automation of Software Test (AST), 2013 8th International Workshop on, pp 111–117

Pan K, Wu X, Xie T (2014) Guided test generation for database applications via synthesized database interactions. ACM Trans Softw Eng Methodol 23(2):12:1–12:27

Pan K, Wu X, Xie T (2015) Program-input generation for testing database applications using existing database states. Automated Software Engg 22(4):439–473

Prud'homme C, Fages JG, Lorca X (2015) Choco Documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., URL http://www.choco-solver.org

de la Riva C, Suárez-Cabal MJ, Tuya J (2010) Constraint-based test database generation for sql queries. In: Proceedings of the 5th Workshop on Automation of Software Test, ACM, New York, NY, USA, AST '10, pp 67–74

Sen K, Marinov D, Agha G (2005) Cute: A concolic unit testing engine for c. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE-13, pp 263–272

Shah S, Sudarshan S, Kajbaje S, Patidar S, Gupta B, Vira D (2011) Generating test data for killing sql mutants: A constraint-based approach. In: Data Engineering (ICDE), 2011 IEEE 27th International Conference on, pp 1175–1186

Suárez-Cabal MJ, Tuya J (2009) Structural coverage criteria for testing SQL queries. J UCS 15(3):584–619

Tsang EPK (1993) Foundations of constraint satisfaction. Computation in cognitive science, Academic Press

Tuya J, Suárez-Cabal MJ, de la Riva C (2007) Mutating database queries. Inf Softw Technol 49(4):398–417

Tuya J, Suárez-Cabal MJ, de la Riva C (2010) Full predicate coverage for testing sql database queries. Softw Test Verif Reliab 20(3):237–288

Tuya J, de la Riva C, Suárez-Cabal MJ, Blanco R (2016) Coverage-aware test database reduction. IEEE Transactions on Software Engineering PP(99), DOI http://doi.ieeecomputersociety.org/10.1109/TSE.2016.2519032

Veanes M, Grigorenko P, Halleux P, Tillmann N (2009) Symbolic query exploration. In: Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering, Springer-Verlag, Berlin, Heidelberg, ICFEM '09, pp 49–68

Vemasani P, Brodsky A, Ammann P (2014) Generating test data to distinguish conjunctive queries with equalities. In: Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation Workshops, IEEE Computer Society, Washington, DC, USA, ICSTW '14, pp 216–221

Willmor D, Embury S (2006a) An intensional approach to the specification of test cases for database applications. In: Proceedings of the 28th International

Conference on Software Engineering, ACM, New York, NY, USA, ICSE ’06, pp 102–111

Willmor D, Embury S (2006b) Testing the implementation of business rules using intensional database tests. In: Testing: Academic and Industrial Conference - Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings, pp 115–126

Zhang J, Xu C, Cheung SC (2001) Automatic generation of database instances for white-box testing. In: Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development, IEEE Computer Society, Washington, DC, USA, COMPSAC ’01, pp 161–165

Zhou C, Frankl P (2011) Inferential checking for mutants modifying database states. 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation 0:259–268, DOI http://doi.ieeecomputersociety.org/10.1109/ICST.2011.63