Coverage-Aware Test Database Reduction

J. Tuya, Member, IEEE, C. de la Riva, M.J. Suárez-Cabal, R. Blanco

Abstract—Functional testing of applications that process the information stored in databases often requires a careful design of the test database. The larger the test database, the more difficult it is to develop and maintain tests as well as to load and reset the test data. This paper presents an approach to reduce a database with respect to a set of SQL queries and a coverage criterion. The reduction procedures search the rows in the initial database that contribute to the coverage in order to find a representative subset that satisfies the same coverage as the initial database. The approach is automated and efficiently executed against large databases and complex queries. The evaluation is carried out over two real life applications and a well-known database benchmark. The results show a very large degree of reduction as well as scalability in relation to the size of the initial database and the time needed to perform the reduction.

Index Terms—Test database reduction, Test coverage of code, Test design, Testing tools

1 Introduction

DATABASE applications involve the management of large amounts of data stored and organized in many tables. These data are usually managed using a third party component called the Database Management System (DBMS) that provides high performance and a high degree of scalability and dependability. The application is able to access the stored data using some kind of query language. Despite the continuous developments in new technologies such as NoSQL databases [1] and persistence systems, applications handling the data using Relational DBMS and the Structured Query language (SQL) [2] are ubiquitous in virtually all industrial and business sectors.

Testing software applications involves a crucial activity that consists of elaborating test cases, each having sets of test case preconditions, inputs and expected outputs [3]. The tester has to provide enough meaningful inputs in order to exercise the application code as much as possible. If the application involves a database, the elaboration of test databases is a determining factor. On some occasions, the test database may be by far the most important component of the input (such as reports, analytical queries or dashboards).

Creating a test database involves a number of technical and practical challenges. The test database should contain enough meaningful data to adequately exercise the application under test. However, populating the test database becomes a difficult task because of the highly interrelated nature of tables. Test databases should be kept small in order to facilitate: 1) the efficiency of the reset of the test database, 2) the fault localization and debugging of failed tests, 3) the test output evaluation when a test produces many outputs from the database, and 4) the maintenance and extensibility of test scripts.

Consider, for example, the following scenario: A database contains orders made by clients. Each order has the information about the client and the warehouse that will supply the goods. This information is stored in a main table (*order*) with the order ID (*oid*), client ID (*cid*), warehouse ID (*wid*) and the order *status*. The warehouse table includes its ID (*wid*) and its *name*. A new reporting module is under development and one of the reports consists in displaying all cancelled orders (*status*='C') and the warehouse name. The developer creates the report based on the following query:

```
SELECT o.oid, o.status, c.cid, w.name
FROM order o, warehouse w
WHERE o.wid=w.wid AND o.status='C'
```

The test requirements for this report include creating test databases with orders with status 'C' and other different statuses. In addition, as the warehouse is assigned after entering an order, there must be orders in the test database that have been cancelled before and after the assignment of a warehouse.

Creating test databases needs a trade-off between the quality of the data from the testing point of view and practical issues related to populating and loading the test database. The tester may adopt different strategies that range from 1) beginning from a previously populated database (e.g. a copy of the production database) to 2) beginning from an empty database.

If testing is done using a production database, the actual results have to be checked over many rows in the report to ensure they meet the specification. In particular it should be checked that all reported rows are included and there are no omitted rows. In this case the query is wrong as it ignores cancelled orders that do not have a warehouse assigned yet. The source of the fault in the query is that the join between tables should be a left join. It should be written as:

```
SELECT o.oid, o.status, c.cid, w.name FROM order o
LEFT JOIN warehouse w ON o.wid=w.wid
WHERE o.status='C'
```

Moreover, if the test is further automated, its execution will require a reset of the database to isolate this test from others that modify the database, which is more time consuming as the size of the database grows.

The second strategy is to start from an empty database. The tester is free to create a script to populate a test database containing only the rows that fulfill the test requirements. The comparison of the actual results is easier as fewer rows at the output have to be checked and the reset of the database is faster. However, the tester has to specify each row and its values (including all columns in the tables involved, which are simplified in the

• *The authors are from the University of Oviedo, Campus Universitario de Gijón, Dpto. Informática, 33204 Gijón, Spain.*
E-mail: {tuya,claudio,cabal,rblanco}@uniovi.es.

example) and to populate additional tables to ensure referential integrity.

An intermediate strategy that constitutes a trade-off between the above would consist of extracting a subset of the data that fulfills the test requirements from the production database and generating a script to populate the test database with this subset. This is a reduction of the production database. If it is made automatically, this would facilitate the testing as it contains few, but meaningful data (that cover the test requirements of the query). It would be easier to check the actual results (they contain fewer rows) and easier to populate and load the test database (the script would be automatically created).

The scope of this paper relates to this intermediate strategy: Given a database, produce a smaller database containing meaningful data to enable its use as a test database. To accomplish this, 1) we start from an initial database (that can be taken from a production database after obfuscating confidential data) and a set of queries that have been issued to the database (which can be taken from the execution log registered by the DBMS). 2) In order to be able to select meaningful test data from the initial database we use a test criterion called SQL Full Predicate Coverage (SQLFpc) [4] which is a variant of Modified Condition/Decision Coverage (MCDC) [5], [6] specifically tailored for SQL. Given a SQL query and a test database, the SQLFpc criterion defines a set of test requirements, each represented as a *coverage rule* (written as a SQL expression). The execution of the rules against the initial database determines whether the test requirements for the query are met. 3) Then the data which satisfy each coverage rule are retrieved, reduced to a subset and inserted into another database (initially empty) which constitutes the reduced test database.

While our prior work on test database reduction [7] deals with tool support for a very limited kind of queries, this work largely expands the applicability of the approach to more complex queries and provides a thorough assessment of the results.

The specific contributions of this work include:

1. The definition of a set of *reduction rules* and *reduction procedures* which allow a) to determine the tables and rows from the initial database that satisfy each test requirement (represented by coverage rules), and b) to select a small set of them in order to guarantee that test requirements are also met in the reduced database. The reduction procedures perform a search on the initial database to select a subset based on cost and fitness functions.
2. The approach is able to handle a *large set of SQL* syntax, including the main clauses (SELECT, JOIN, WHERE, GROUP BY, HAVING) as well as subqueries and views.
3. Several *optimization strategies* allow decreasing the total time needed to reduce the database by parallelizing different tasks and reducing the number of rows that need to be retrieved from the database.
4. The reduction *preserves the coverage* in most cases, the final size of the reduced test database is generally *independent* from the size of the initial database and the approach is *scalable*. This has been checked in three case studies (two of them are taken from real-life applications and the third from a synthetic benchmark).

The remainder of the paper is structured as follows: Section 2 provides background and related work on test reduction, relational models, database testing as well as the basic notation and the concept of coverage. Section 3 formulates the database reduction problem. Section 4 presents how to elaborate the reduction rules and procedures for different types of queries and Section 5 deals with optimization issues. Section 6 evaluates the approach and Section 7 discusses the main results. Finally, Section 8 concludes.

2 Background and Related Work

2.1 Test Suite Reduction

A number of different approaches have been studied in the past in the field of regression testing to maximize the value of a given test suite: minimization, selection and prioritization [8]. 1) Test suite minimization seeks to eliminate redundant test cases in order to reduce the number of tests to run, 2) Test case selection seeks to identify the test cases that are relevant to some set of recent changes, and 3) Test case prioritization seeks to order test cases in such a way that early fault detection is maximized.

Test suite minimization is often called test suite reduction. Rothermel et al. [9] formulate the problem as follows:

Given: Test suite T, a set of test-case requirements $r_1, r_2, \ldots, r_n$ that must be satisfied to provide the desired test coverage of the program, and subsets of T, $T_1, T_2, \ldots, T_n$, one associated with each of the r_i's such that any one of the test cases t_j belonging to T_i can be used to test r_i .

Problem: Find a representative set of test cases from T that satisfies all r_i's.

Many algorithms and empirical studies related to the above test regression problems have been published during the last few decades, as well as some comprehensive reviews [8], [10]. One of the most important concerns on test suite reduction is whether a minimized test suite preserves the fault detection ability that the original test suite had. In this sense the empirical studies have not been conclusive. Wong *et al*. reported a decrease in fault detection ability of 1.45% in the worst case using 10 Unix programs [11] and 7.28% using a larger real-life program (space) [12]. However other studies found different figures. Rothermel et al. found a decrease in fault detection ability of over 50% [13] using the Siemens Suite. Using the space program they found 8.9% [14]. If the test suites are randomly generated the decrease is larger (18.1%). These figures depend on a great number of factors such as the programs themselves, how the test cases and test suites have been obtained and the kind of faults that are considered [8].

In this paper we deal with a similar problem to the test suite minimization, but, instead of obtaining a reduced test suite, we seek to obtain a reduced test database.

2.2 The Relational Model

The relational model was first developed by Codd [15] and defines the foundations of data storage and querying that is implemented in today's commercial relational database management systems. The notation used in this paper is that presented by the author in the second version of the relational model [16], referred to as RM/V2, with some adaptations needed for subsequent sections.

Relations and attributes: Given a set A of attributes $A_1, \ldots, A_m$, a relation R is a subset of the Cartesian product of their domains, denoted as $R(A_1, \ldots A_m)$ or simply $R(A)$ or R. In other words, a relation $R(A)$ is a set of tuples of the attributes in A. In SQL a

relation is a table or view, attributes are columns and tuples are rows. For each relation one or more attributes are *primary keys* which uniquely identify each tuple in this relation. The domain of attributes includes special marks to reference missing or inapplicable attributes which are indicated as NULL in commercial relational DBMS. This leads to a three-valued logic of predicates.

The basic operations in the RM/V2 transform either a single relation or a pair of relations into another relation. Operators are defined using relational assignments: A *relational assignment* is in the form $Z \leftarrow rve$ where *rve* denotes a *relation-valued expression* (RVE) and Z is the name of the relation obtained when applying the relation-valued expression. In SQL an RVE is called a query. The basic *relational operators* are shown below:

Selection: Select operator $Z \leftarrow R[p(A)]$ generates as a result a relation Z which contains the complete tuples from relation R that fulfill the predicate p on attributes A. Its SQL expression is:

SELECT * FROM R WHERE p(A)

Projection: Project operator $Z \leftarrow R(A')$ generates a relation Z which contains only the subset of attributes specified in $A' \subset A$. Its SQL expression is:

SELECT A' FROM R

Joins: The join operator $Z \leftarrow R[p(A,B)]S$ generates a relation that contains tuples of $R(A)$ concatenated with tuples of $S(B)$, but only where the condition specified by predicate $p(A,B)$ is found to hold true. The predicate p is named *join predicate*. Its SQL expression is:

SELECT * FROM R INNER JOIN S ON p(A,B)

The above is also called inner join. When the outer increment (tuples from a relation that do not fulfill the join predicate) is added to the resulting relation, the join is called outer join. Depending on the relation that is considered in the outer increment (R, S or both) the join is called left outer, right outer or full outer join, respectively. These joins will be denoted as $R[p(A,B)]^{JT}S$, where JT is a label {*L, R, F*} that denotes the join-type.

Framing: The framing operator $Z \leftarrow R /// G$ partitions a relation into a collection of subrelations (groups), such that each of them has equal values for a set of attributes G named *grouping attributes*. The most commonly used is in the form $Z \leftarrow R /// G(G,F)$ which performs aggregated calculations over all tuples on each frame. These calculations are performed by *aggregate functions* (denoted by F). Its SQL expression is:

SELECT G,F FROM R GROUP BY G

A further select operator may be applied after framing: $Z \leftarrow R /// G [q(G,F)] (G,F)$. Predicate $q(G,F)$ involves grouping attributes and aggregate functions over A. It is called *frame predicate*, which is represented in SQL by the HAVING clause:

SELECT G,F FROM R GROUP BY G HAVING q(G,F)

2.3 Test Coverage for Database Applications

Test coverage criteria for database applications include fault-based and flow/logic-based. In the fault-based category the existing works range from the development of sets of mutants for SQL queries [17], [18] or schemas [19], [20], [21] to the evaluation of the fault-detection effectiveness with tools [22], [23], [24], [25], fault-localization [26] and empirical studies [27]. Others are application specific, mainly with the goal of detecting SQL injection vulnerabilities [28], [29], [30] and preventing them [31].

In the flow/logic-based category some criteria are based on data-flow [32], [33] as well as tools to automate the approach [34]. These criteria have also been used in the context of active databases [35]. Logic-based criteria incorporate a notion of multiple condition coverage [36], [37], define a hierarchy of criteria to test schema constraints [38] or focus on how the SQL strings containing the query to be executed are constructed by the program [39]. The SQLFpc criterion [4] mentioned in the introduction belongs to this category. As this paper will make use of this criterion the rest of this subsection provides a brief summary.

MCDC [5], [6] is a coverage criterion that specifies test requirements consisting in that every condition in a logical decision has taken all possible outcomes at least once, and each condition has been shown to independently affect the decision's outcome. It is also called Active Clause Coverage [40][41]. Based on this principle, SQLFpc provides a criterion tailored for the specific features of SQL, where the test inputs are the database and the programs are SQL queries. In addition to conditions 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 a SQL query, SQLFpc specifies a number of test requirements that impose a set of constraints on the test database in order to achieve the coverage, which are called *coverage rules* (Δ). Coverage rules are obtained by applying *coverage rule transformations* (Φ) to the query [4] and are expressed as SQL queries. The evaluation of SQLFpc coverage against a previously populated database is obtained by executing each rule (query) against the database. If the output is not empty (it obtains at least one row), it means that the test requirement embodied in the rule is fulfilled. The approach for reduction presented in this paper will take these outputs and will try to obtain a subset of rows from the initial database that produces a non empty output.

2.4 Testing Database Applications

Most of the previous work on testing database applications focuses on generating either the sets of test inputs or the test database. The AGENDA tool [42], [43], [44], [45] is loosely related to the category-partition method and uses heuristics to fill the test database. Different criteria may be used to populate the databases. Some of them may take the form of intensional specification by specifying constraints in a SQL-like language [46], [47] or be based on reverse query processing [48] which uses the desired output as extensional specifications to generate the inputs or SQL rules in the form of intensional specifications [49].

Other approaches directed towards test data generation use different kinds of constraint solvers to either generate test inputs or populate the test database. With the goal to test SQL queries, some works address the test database generation by imposing constraints on the database statements using different test criteria, such as predicate coverage [50], [51], [52], mutation coverage [53], [54] or the SQLFpc criterion [55]. Constraint-based approaches are also explored to test database programs. These approaches 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 [56], [57] or both program inputs and test databases [58], [59], [60]. Search based techniques have also been used to generate test databases considering schema constraints [61].

Although most of the existing work deals with the generation of test cases, other issues such as debugging and regression testing have also been handled. These include the reproduction of

problems [62][63], removing redundant tests [64], obtaining the execution order of a given test suite in order to reduce the number of resets of the test databases [65], [66], [67] or the selection of test data for regression [68], [69], [70]. While these regression approaches are typically white-box oriented, as they take knowledge from the internals of the applications, other black-box approaches [71], [72] perform test selection based on a Classification Tree Model used as specification. The closest work to our work [73] shares our goal to obtain a reduced database from an initial database. Given a set of single-table SQL queries and a fixed size of the reduced database, after generating a set of mutants [17] for all queries, it performs a search on the initial database using genetic algorithms. The result is a reduced database that has a similar mutation score to that of the initial database.

Reverse query processing [48] generates the inputs that produce a given output for a query. A related problem is to obtain the existing inputs that produce a given output, which is called data provenance or lineage [74][75][76]. This has a number of applications that range from scientific workflows [77] to tracing data warehouse transformations [78] or Big Data [79]. The classification of different approaches depends on how we define that an input contributes to the output (contribution semantics). An initial classification is between *why* and *where* provenance [80]: the former refers to the source data tuples that had some influence on producing the output, while the latter refers to the location in the input (attributes) from which the data is extracted; refer to [81] for a comprehensive classification and overview of different approaches.

Data lineage has been computed by defining a set of tracing queries over the source data of data warehouses [82][83]. The closest to our work is the Perm System [81][84][85][86] that extends the PostgreSQL DBMS and rewrites queries to obtain a provenance query that determines the source data. The reduction rules described in this paper also compute the source data for the coverage rules, but the implementation does not depend on a given DBMS and adds additional information needed to allow the reduction procedures to select a subset of the source data.

This article shares some goals with most of the aforementioned works although it also pursues additional objectives: 1) to generate a test database (reusing an existing database instead of generating its data from scratch), and 2) to reduce the size of the tests (reducing the size of the test database instead of reducing the size of the test suite).

3 Problem Statement

Section 2.1 formulated the general problem of test suite reduction. It relies on a set of test requirements that must be satisfied by both the original and the reduced test set. Usually, these test requirements are stated in terms of some structural test coverage criterion (e.g. individual decisions made by branches in the procedural code). If the program uses a database, part of the decisions taken in the code depend on the result of queries executed against the data stored in the database. Therefore, part of the application logic is embedded in the SQL queries that access the database. The extreme case is an application intended for reporting, in which most of the logic relies on the SQL instead of the procedural code. In this case, the goal is to reduce the size of the test data, instead of the number of test cases. The ideal situation would be to include test requirements related to both the queries and the procedural code of the program, and then perform the reduction of the test suite and the amount of data stored in the test database. This paper focuses on the second of the aforementioned cases: the reduction of the test database.

The output produced by a SQL query depends on the decisions taken in the query (e.g. WHERE clauses) as well as other clauses (e.g. the different types of joins). The SQLFpc coverage criterion [4] defines a set of test requirements to exercise a query. Each requirement is also represented by a SQL query (coverage rule). When a coverage rule is executed against a given database, it produces a number of rows. Each row has been obtained from a subset of data that fulfills the test requirement (i.e. covers the rule). Therefore, it is enough to obtain a single row at the output to ensure that the test requirement is met. This will be the basis for the database reduction as if we obtain the subset of the database that produces a single row at the output of the coverage rule, this subset will constitute a reduced database for that test requirement.

In terms of the scope of this paper the reduction problem is focused on the database and the queries that are executed by a program. In this problem, the test suite is the set of tuples in a database, the program is a set of SQL queries and the test requirements are determined by a set of coverage rules obtained by applying the SQLFpc coverage criterion (Section 2.3) to the queries under test. Taking into account the above considerations, the reduction problem is reformulated as:

Definition 1 (Test Database Reduction Problem). Given: An initial database D, a set $\{Q_j\}$ of RVEs (SQL queries), a set of SQLFpc coverage rules $\{\Delta_i\}=\cup\Delta(Q_j)$ that are covered when evaluated over D where $\Delta(Q_j)$ denotes the set of coverage rules for Q_j, and the subsets of D, D_1, D_2,… D_n, one associated with each of the Δ_i's such that any one of the D_i covers the rule Δ_i.

Problem: Find a reduced database D' which contains a representative set of tuples from every D_i such that tuples from D_i cover the rule Δ_i and every coverage rule Δ_i is covered when evaluated over D'.

Coverage gain and loss. The test suite reduction problem (Section 2.1) and the test database reduction problem (Definition 1) have a significant difference: Test cases T_k in test suite reduction are independent. Adding a test case never decreases the requirements coverage. However, subsets D_i in test database are not independent. A single tuple may be shared by several D_i causing a situation where adding tuples could uncover some requirement previously covered (coverage loss). For example, if a requirement $sum(a)<3$ is covered and then we try to cover $sum(a)\geq 3$, the first requirement will become uncovered when adding tuples to cover the second one. The contrary may also occur (coverage gain), for example, in the case of outer joins. These issues will be discussed in further sections.

The primary purpose of the test database reduction described here is to provide a starting point of a test database in order to facilitate the reset of the test database, the fault localization and debugging of failed tests, the test output evaluation and the maintenance and extensibility of test scripts. This will help the tester to develop, complete or automate functional tests.

Therefore, the purpose is to achieve a reasonably good solution in terms of coverage and size of the reduced database, although coverage may not be exactly the same as the initial database for a few rules.

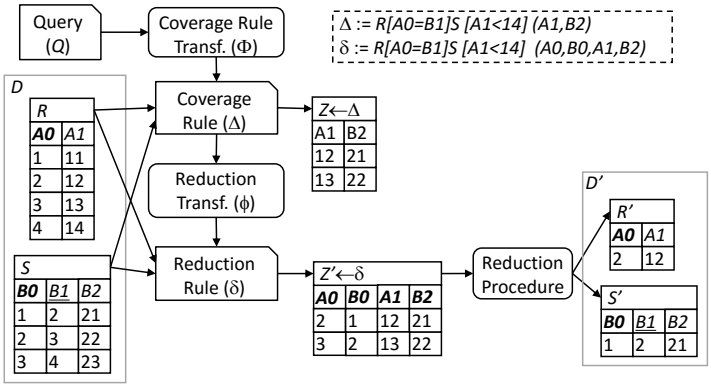

Figure 1. Example of database reduction for simple queries with joins

Approach: In general, the approach to produce a reduced database is based on:

1. The definition of *reduction transformations* (ϕ) that produce *reduction rules* (δ) from the coverage rules (Δ). Reduction rules are RVEs that compute the provenance of the coverage rules when evaluated over the initial database.
2. The definition of *reduction procedures* for selecting a small number of tuples from those obtained by the evaluation of the reduction rule. These tuples are going to be added to the reduced database.

After executing the reduction procedures for each reduction rule δ, the set of all resulting tuples plus additional tuples needed to enforce the referential integrity are copied to the reduced database (initially empty).

4 Reduction Rules and Procedures

This section details how the coverage rules are transformed into the reduction rules and how tuples resulting from the evaluation of the reduction rule are selected to obtain the reduced database. Firstly, we cover the most basic operators (Section 4.1), followed by frames (Section 4.2) and subqueries (Section 4.3). Finally, we summarize how all of them are combined (Section 4.4).

4.1 Join and Select Operators

The case of an RVE (query) that performs joins and a further selection of joined tuples is the simplest one and allows the presentation of the foundations of the reduction approach. We first illustrate it with an example.

Example 1. Consider an initial database D={R,S} which contains two relations $R(A0,A1)$ and $S(B0,B1,B2)$, being $A0$ and $B0$ primary keys and an RVE Δ which represents a coverage rule defined as:

$$\Delta := R[A0=B1]S\ [A1<14]\ \ (A1,B2)$$

In SQL: SELECT A1, B2 FROM R INNER JOIN S ON R.A0=S.B1 WHERE R.A1<14

The reduction problem as stated in Section 3 consists in selecting a subset $D'\subset D$ (i.e. subsets $R'\subset R$ and $S'\subset S$) such that Δ is covered when evaluated over D' (reduced database).

Figure 1 depicts the global approach for this example: Starting from a query Q a coverage rule Δ is obtained by applying a coverage rule transformation Φ [4]. By evaluating the coverage rule Δ over D, a relation Z is produced. The coverage rule is covered iff Z contains at least one tuple (as this means that there is some subset of data that satisfies the requirement specified by the coverage rule). Therefore, every subset D'={R', S'} that produces at least one tuple of Z after evaluating the coverage rule Δ over D' is a reduced database. These subsets are obtained by executing a reduction rule and a reduction procedure as shown below.

Reduction transformation and reduction rule. A reduction transformation (ϕ) transforms an RVE (coverage rule) into a reduction rule (δ) in such a way that relation Z' obtained after evaluating δ over D allows identifying all source tuples of D. In the example, this is accomplished by including the primary keys in the projection. The result is a new RVE called reduction rule.

$$\delta := R[A0=B1]S\ [A1<14]\ (A0,B0,A1,B2)$$

In SQL: SELECT A0,B0,A1,B2 FROM R INNER JOIN S ON R.A0=S.B1 WHERE R.A1<14

Definition 2 (Reduction transformation for joins with select). Let Δ := $R[p(A,B)]^{JT}S[q(A,B)]$ be a coverage rule which joins tuples in relations $R(A)$ and $S(B)$ (called *source relations*) and then selects some of the resulting tuples according to a predicate q. The reduction transformation is the same as Δ including a projection on all attributes A,B:

$$\phi_{JT}(\Delta) := \Delta\ (A,B) \quad (1)$$

Then the reduction rule is $\delta=\phi_{JT}(\Delta)$. Primary keys of all relations in δ are called *source keys*.

Reduction procedure. A reduction procedure selects a small subset of tuples of $Z'\leftarrow\delta$ and finds the source tuples in D to obtain the reduced database D'={R', S'}. The reduction procedure must use some kind of strategy in such a way that the reduced database is as small as possible. This is based on the cost of adding each tuple of Z' to D' measured in terms of the number of new tuples that have to be added to the reduced database. In Example *1*, as can be seen in Figure 1, Z' contains two tuples. Either of them will produce tuples in R' and S' with cost 2. Each of them may be selected (for example, the first tuple of Z' in the figure). The reduction procedure is incremental, being executed for each coverage rule, so that it takes into account tuples that are already in the reduced database. For example, if a tuple with $A0$=3 already exists in D', the cost of the second tuple in Z' will be 1 (a lower cost) as only one new tuple with $B0$=2 would need to be added to the reduced database.

Definition 3 (Reduction procedure for joins with select). Let {δ_i} be a set of reduction rules, D={R_j} the initial database and D'={R_j'} the reduced database (initially empty). Let z_i^k be the kth tuple of the relation obtained after evaluating $Z'_i\leftarrow\delta_i$ and $source(z_i^k)$ the set of tuples in D that are determined by the primary key values that are in z_i^k (source relation). Let $cost(z_i^k,D,D')$ be an integer expression which counts the number of tuples in $source(z_i^k)$ that are in D, but are not in D'.

The reduction procedure follows the algorithm in Figure 2: Each coverage rule Δ_i is transformed into the reduction rule δ_i and evaluated against the initial database D. Then for each tuple z_i^k the set of source tuples producing the minimum cost is considered the best solution and added to the reduced database D'.

Coverage gain and loss: As the reduction procedure only adds tuples to the reduced database at each step, when a query has only selection and join operators, there are no coverage rules that are covered at some step and become uncovered at a later step (coverage loss). However, the opposite is not true: a rule that is not covered in the initial database may become covered

Let $D'=\emptyset$ (reduced database initially empty)
For each coverage rule Δ_i of every query
  Let *BestDB*=∅; *BestCost*=∞ (initial cost)
  Let $\delta_i = \phi(\Delta_i)$ (obtain the reduction rule)
  Let $Z'_i \leftarrow \delta_i$ (evaluation over the initial database)
  For each z_i^k in Z'_i
    Let *CurrentDB*=*source*(z_i^k) (candidates to insert in D')
    Let *CurrentCost*=*cost*(z_i^k,*D*,*D'*) (cost of adding z_i^k to D')
    If *CurrentCost*=0 (no cost, no data to add)
      BestDB=∅; exit inner loop
    If *CurrentCost*<*BestCost* (cost improves, save best solution)
      Let *BestDB*=*CurrentDB*; *BestCost*=*CurrentCost*
  Let $D'=D' \cup BestDB$ (add the best solution found to D')

Figure 2. Algorithm to select the reduced database with lowest cost

in the reduced database (coverage gain). This is the case of rules with outer joins: Initially, master relations may have at least one related row in their detail, so, rules that require a master without any detail are not covered by the initial database. However, as the reduction process selects only a few rows from the initial database, the reduced database may contain situations in which there is a row in a master table without any related row in the detail, leading to a coverage gain.

4.2 Framed Relations

Consider a coverage rule Δ that contains a framing operator:

Δ := *R///G [q(G, F)] (G,F)*

In SQL: SELECT G,F FROM R GROUP BY G
HAVING q(G,F)

Where the grouping attributes *G* are attributes of *R* and *F* are aggregate functions on the attributes of *R*. The approach is illustrated below with an example.

Example 2. Consider an initial database *D*={*R*} with a single relation *R(A0,A1,A2)*, being *A0* the primary key, and a coverage rule expressed by the following RVE:

Δ := *R///A1[sum(A2)>6] (A1,sum(A2))*

In SQL: SELECT A1,SUM(A2) FROM R GROUP BY A1
HAVING SUM(A2)>6

Figure 3 depicts the global approach for this example, which is described below.

Reduction transformation and reduction rule. As the framing hides the tuples and primary keys that are grouped in each frame, the first step is to ungroup the frame by joining the relation *Z* obtained by evaluating the coverage rule with the original relation *R* using the grouping attributes (*A1*) as the joining attributes, and then ordering by the grouping attributes of *Z*. The resulting relation (*Z'*) reveals the frames.

Definition 4 (Reduction transformation for frames with select after frame). Let Δ := *R///G[q(G, F)]* be a coverage rule which frames the relation *R* according to the grouping attributes *G* and then selects those that fulfill the predicate *q(G,F)*, where *F* are aggregate functions over attributes in *R*. The reduction transformation generates an RVE that joins the coverage rule Δ with R based on the grouping attributes:

$$\phi_{FS}(\Delta) := \Delta\ [\Delta.G_1=R.G_1 \wedge \Delta.G_2=R.G_2 \wedge \dots\ ]\ R \qquad (2)$$

Then the reduction rule is $\delta=\phi_{FS}(\Delta)$. Relation Δ is called *group relation* and relation *R* is called *source relation*. The grouping attributes are called *group keys*. The primary keys of *R* (*source keys*) determine tuples that form the group. The reduction rule is ordered by the group keys *G* in order to be able to perform the reduction using a sequential exploration of tuples retrieved. If

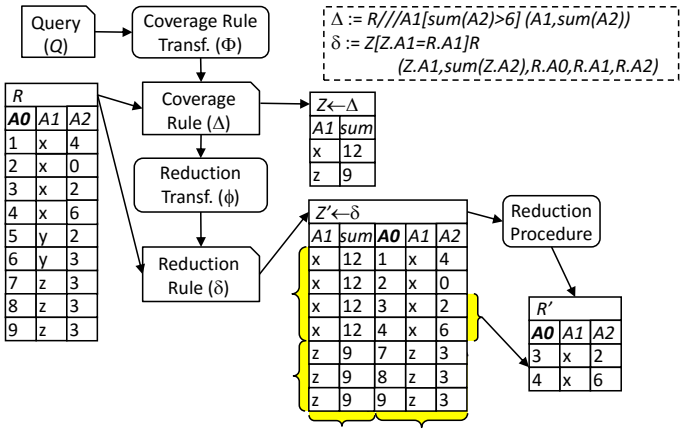

Figure 3. Example of database reduction for queries with frames and select after frame

relation *R* is the result of evaluating a query with joins and select operators, it is first transformed as indicated in Section 4.1.

Reduction procedure. A naïve strategy would consist of applying the reduction procedure described in Section 4.1 considering r_i^k as a subrelation containing all tuples in each frame of *Z*, and then selecting that with the lowest cost. However, each frame may contain thousands or millions of tuples, and then the reduction may be considerably far from the optimum. Selecting an arbitrary subset in each frame is not possible because the candidate tuples must fulfill the constraint stated by the frame predicate (*sum(A2)*>6 in the example). Therefore, each frame must be reduced before checking the cost of adding its source tuples to the reduced database. Another practical constraint is that the reduction must be able to handle frames containing many rows that may exceed the memory available, so that the reduction will explore the tuples in frames on the fly (in the order that they are obtained from the database).

The reduction of each frame will be made using a greedy algorithm that sequentially explores each tuple in a frame and adds it to the reduced frame if it improves a fitness function. This function measures the distance of a candidate solution from fulfilling the frame predicate.

The reduction procedure uses the algorithm described in Section 4.1 with two differences: 1) z_i^k is the set of tuples that compose a frame, instead of a single tuple, and 2) the set z_i^k is to be reduced as much as possible while fulfilling the frame predicate *q(A, F)*. To do this we use a search algorithm before the evaluation of the cost.

Reduction of frames. Search based algorithms have been previously used for a number of software engineering problems [87][88] and in particular, software testing [89][90]. A fundamental issue is the definition of a fitness function that is minimized in order to find the best solution among a number of candidate solutions.

The fitness function incorporates a notion of distance or cost that measures how far a candidate solution is (i.e. a set of inputs) from satisfying some criterion (e.g. make true or false a given condition or decision). The distance for relational expressions is evaluated using cost functions, such as that defined by Tracey et al. [91]. Fitness functions usually include the cost and an additional term called approach level [92].

When ranges of variables are very different, using a simple

```
Let X'=∅
For each subrelation T⊂X composed by a single tuple
(candidate tuple)
  If processing first iteration then
    Let X'=T
  Else
    Let f=fitness(q,T, X') (fitness caused by adding T to X')
    If f>0 then
      Let X'=X'∪T (add tuple contained in T to reduced frame)
    If f=1 then (condition is true, solution is found)
      Try to remove every single tuple in X' whenever
       fitness is 1 after removal and return
```

Figure 4. Algorithm to reduce a frame

distance function may distort the results. Consider the decision a>5 AND b<2020, where the ranges of a and b are 0..10 and 0..10000, respectively, and two pairs of inputs a=0, b=1910 and a=4, b=1900. If the distance of a logical expression is defined as the sum of the individual distances, the distance of the second pair is larger by 6 units. However, in this context the second pair is a better solution because the value of the first condition is very close to the solution. Several approaches to normalize the distances are given in the literature such as that defined by Baresel [93] which expressed the normalized distance as $1\text{-}1.001^{-dist}$. However, this kind of normalization has some drawbacks [94].

The problem of searching for solutions in the context of the reduction of frames, which is the subject of this section, has some particularities that have to be handled:

1. A solution is a set of tuples (a subset of the frame that has to be reduced). Expressions contain aggregate functions over attributes of the solution that have to be evaluated over sets of tuples.
2. The problem does not consist of generating new candidate tuples, but rather of selecting a set of tuples from the original frame that will be included in the reduced frame.
3. The algorithm is constrained by the practical necessity of a sequential evaluation of the original tuples, which may contain a large number of tuples that cannot fit in the memory. Therefore, each one of the visited tuples (candidate tuple) has to be considered on the fly either to be added or discarded into the reduced frame.
4. The normalization of distances is very important, because logical expressions may involve relational expressions including variables with short ranges (e.g. when counting the number of tuples in the frame) and variables with large ranges (e.g. when adding currency values).

Taking into account the above considerations, the definitions of distance, fitness and reduction procedure for frames are provided below:

Distances for base predicates: Let $X \subset Z'$ be a single original frame that is being reduced and X' the reduced frame (initially empty). Let p_i be a base predicate, which may contain references to attributes or aggregate functions over attributes but not logical expressions. The distance $d(p_i, X')$ over the relation X' is calculated using the Tracey functions [91].

Consider, for example, a predicate p:=$sum(a)$≥8 evaluated over a relation with three tuples {(1), (2), (3)}. The evaluation of the distance calculates the term $sum(a)$ which gives 6. Then the distance is 8-6=2.

Definition 5 (Fitness function for the frame predicate). Let T be a subrelation $T \subset X$ (candidate tuples to be added to X'). The fitness $f(p_i, T, X')$ of predicate p_i when adding candidate tuples in T to X' is calculated with respect to the previous distance (before adding the tuples T) according to:

$$fitness(p_i) := f(p_i, T, X') = 1 - \frac{d_{cand}}{d_{prev}} = 1 - \frac{d(p_i, X' \cup T)}{d(p_i, X')}$$

where d_{prev} is the distance of the predicate p_i with regard to the relation X' and d_{cand} is the distance of the predicate p_i with respect to relation X' after adding the candidate tuples T.

Note that this definition of fitness is relative to the distance of a previous solution and therefore it prevents the aforementioned normalization problem. A positive value of fitness means that adding the candidate tuple approaches the reduced frame to the solution and conversely for negative values. A value of 0 means no change and a value of 1 means that a solution has been found.

Let q be a predicate based on a logical expression over predicates p_i. The fitness function for logical expressions is calculated as:

$$fitness(p_1 \wedge p_2) := average(fitness(p_1), fitness(p_2))$$
$$fitness(p_1 \vee p_2) := maximum(fitness(p_1), fitness(p_2))$$

Note that these functions are different to the Tracey's functions: For AND operators fitness of p_1 and p_2 cannot be added because the fitness is upper bounded by 1. For OR operators the maximum is calculated instead of the minimum because in this case larger values of fitness mean lower distance to the objective.

Definition 6 (Reduction procedure for frames). The reduction procedure for a frame X and a frame predicate q follows the algorithm in Figure 4.

Let us return to the example shown in Figure 3 to illustrate how the frame reduction works. The reduction rule produces two frames (for values x and z of $A1$). When the above algorithm processes the first frame it performs four iterations, one for each tuple:

1. The first tuple is added to X'.
2. d_{prev}=6-$sum(A2)$+ε=2+ε. The second tuple adds a value of 0 for $A2$. Then d_{cand}=d_{prev}, therefore, $fitness$=0. As fitness is not positive the tuple is skipped.
3. d_{prev}=2+ε. The third tuple adds a value of 2 for $A2$, so $sum(A2)$=6 and d_{cand}=6-6+ε=ε, therefore, $fitness$=1-ε (the predicate $sum(A2)$>6 still is false). This tuple is added to X'.
4. d_{prev}=ε. The fourth tuple adds a value of 6 for $A2$, so $sum(A2)$=12 and d_{cand}=0, therefore, $fitness$=1 (The predicate is fulfilled) and this tuple is added to X'.

Once X' is obtained, it is explored again in order to check whether some tuple may be removed while keeping the predicate $sum(A2)$>6 true. Removing the first tuple fulfills this predicate so that it is removed from X'. No other tuples fulfill this predicate, so that solution is composed by the 3rd and 4th tuples of the original frame.

The same algorithm is applied to the second frame, which gives a solution consisting of all tuples (1st, 2nd and 3rd tuples).

From the above reduced frames the first one is selected as its cost is 2 (it adds two tuples) while the cost of the second reduced frame is 3 (it adds three tuples).

Note that the selection of the frame to be included in the reduced database depends on the initial state of the reduced database. In this example an empty reduced database was assumed

as starting point. However, if the reduced database before the algorithm contained the last two tuples of X', the selected frame would be the latter, because, although it consists of three tuples, only one of them would be added to the database.

The implementation of the reduction algorithm supports the SQL aggregate functions *count*, *max*, *min*, *avg*, *sum*. When aggregate functions are involved in a predicate their arguments are included as additional attributes in the reduction rule. The remaining scalar subexpressions are also converted into additional attributes in order to facilitate the evaluation and to support SQL functions. When a frame is evaluated to determine the fitness function the predicate is algorithmically evaluated (logical, relational and arithmetic operators as well as null value semantics are supported). The aggregate functions are evaluated over the tuples in the frame using the attribute representing its argument (which will take different values at each tuple). Although the RVE's in the relational algebra are set oriented, SQL is bag oriented. Therefore, by default duplicates are not removed in SQL unless the distinct set quantifier is specified. The evaluation of aggregate functions takes into account this semantics and includes support for the distinct quantifier in aggregate functions.

Coverage gain and loss: There are some occasions on which a rule that is covered becomes uncovered after adding additional rows. Two main kinds of coverage rules may produce these coverage losses: 1) A rule with a frame and a select after the frame: If the reduction algorithm is unable to obtain a solution for any frame, this rule will lose the coverage (due to the greedy and sequential design of the reduction procedure). 2) A rule with frame predicates that involve aggregate functions: For instance, during reduction a frame predicate $sum(a)<10$ may be fulfilled, but after applying the reduction algorithms to other rules, new rows with large positive values for a may be inserted in the same reduced database.

In the second case, the problem is unsolvable for pairs of rules without grouping attributes in the form $sum(a)<10$, $sum(a)\geq 10$ because they cannot be simultaneously true in the same database. It can be potentially solvable if frames have grouping attributes by ensuring each rule is satisfied in different frames. With the current implementation, as the reduction procedures process each rule sequentially, there is a possibility that the same frame be selected leading to a coverage loss. This could be avoided by reprocessing these uncovered rules to ensure that they are satisfied over different frames.

4.3 Subqueries

Reduction transformations for subqueries are inspired in earlier works from Kim [95] and Ganski and Wong [96]. These are motivated by the difficulty of DBMSs to evaluate efficiently the subqueries. Basically, they consist of removing the subqueries by joining the subquery expressions in the main query. In this article the goal is different as the problem is to obtain the source tuples that are processed in the evaluation of a reduction rule to produce an output, but the key ideas of these transformations are still applicable.

Consider an RVE in the form $R[p]$ where R is a relation (which may contain joins) and p is the select predicate. Select predicates with subqueries include expressions on attributes A of R and subqueries S. There are three different kinds:

- *Scalar subquery*: A *rop* (S), where *rop* is a relational operator.
- *Logical predicates*: A [not] in (S), [not] exists (S).
- *Quantified comparison predicate*: A *rop* [[not] any | some | all] (S).

Additionally, depending on the form of S there are other variants:

- *Correlated subquery*: where a select expression in S references some attribute on the outer query.
- *Group subquery*: where S includes grouping and aggregate functions.

The reduction transformation creates a reduction rule that joins relation R with S, over a predicate for joining only the tuples that satisfy the subquery expression as shown below:

Definition 7 (Reduction transformation for scalar subqueries). Let $\Delta := R[p]$ be a coverage rule, where p contains some scalar subquery expression in the form A *rop* $S[q]$ (B_0). Let B_0 be the (unique) projected attribute of S. Let *replace(x,y,z)* be the replacement of y by z in x. The reduction transformation is defined as:

$$\phi_{SQ}(\Delta) := R[A \text{ rop } B_0]^L S\ [replace(p,S,B_0 \wedge q)] \quad (3)$$

In SQL: SELECT * FROM R LEFT JOIN S ON A *rop* B_0 WHERE replace(p,S,B_0 AND q)

Then the reduction rule is $\delta=\phi_{SQ}(\Delta)$. Relation R is called *base relation* and relation S is called *source relation*. Source tuples are determined by the primary keys of both R and S.

Example 3. Consider the following RVE (coverage rule):

$$R[A1=2 \vee A2 = (S[B1=0](B0))$$

In SQL: SELECT * FROM R WHERE A1=2 OR A2=(SELECT B0 FROM S WHERE B1=0)

Using the previous definition, the reduction rule is:

$$R[A2=B0]^L S\ [A1=2 \vee (A2=B0 \wedge B1=0)]$$

In SQL: SELECT * FROM R LEFT JOIN S ON A2=B0 WHERE A1=2 OR (A2=B0 AND B1=0)

Note that join type is left and the join predicate has also been added to the select predicate. This allows handling OR expressions that may be true even if the subquery expression is false.

The other kinds of subqueries are handled as particular cases:

Case 1 (in, not in). A [not] in S is replaced by $A=S$ and $A\neq S$, respectively before applying the reduction transformation.

Case 2 (exists, not exists). For exists subqueries A *rop* B_0 is replaced by true in the reduction transformation. In the case of not exists, the transformation is simplified to the original query $R[p(A)]$ as no tuples have to be selected from S.

Case 3 (any, some, all subquery). Predicates are replaced by their equivalents using scalar subqueries.

Case 4 (group subquery). The RVE of the subquery includes groups in the form $S[q]//G[r]$ $(aggr)$, where *aggr* is an aggregate function over the attributes of S. To obtain the reduction transformation, the subquery is first transformed according to Definition 4 and then Definition 7 is applied. To ensure that the reduction process finds only tuples that satisfy the subquery condition a *correlation frame predicate* is created in the form A *rop aggr*. If a subquery has groups, the frame predicate is the logical conjunction of these and that obtained by the group transformation.

Case 5 (correlated subquery without groups). In a correlated subquery S, the select predicates of relation S contain references to attributes of the parent relation R. As the general transformation includes the select predicate of subquery S, no additional considerations are needed.

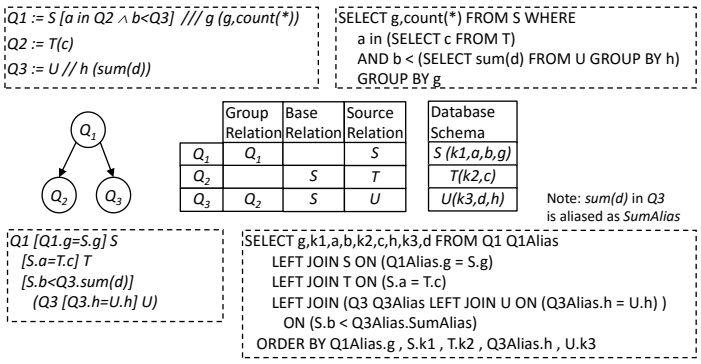

Figure 5. Example of nested queries and their reduction rule

Case 6 (correlated subquery with groups). When a correlated subquery includes groups it is first transformed according to Kim's JA Nested query algorithm [95].

Reduction procedure. When a subquery does not contain groups, only a tuple is needed to satisfy the coverage criterion. Therefore, the reduction procedure is that of queries with select and joins (Section 4.1). If a subquery contains groups, the reduction procedure is that of queries with frames (Section 4.2).

4.4 Reduction for Combinations of Operators

The previous section deals with coverage rules that include a single nesting between two relations. In general, a coverage rule may contain different combinations of operators. Figure 5 displays at the top an example of a coverage rule with a frame operator and two subqueries, the second one also includes frames. The generation of reduction rules and the reduction procedure in these cases is described below and illustrated with an example.

Reduction rules. Given a main RVE which contains nested RVEs, the generation of its reduction rule δ proceeds recursively (depth first) starting from the main RVE. Given a current RVE that is visited:

- The appropriate transformation rule is applied to remove the nesting or frame. If the RVE is a subquery containing a frame, the transformation of the frame is performed before the transformation of the subquery.
- A set of attributes, called *frameset* is added as a projection in the reduction rule. These attributes include the grouping attributes (if any), the base and source keys and other attributes from source relations needed to evaluate expressions (e.g. arguments in aggregate functions).

The reduction rule is ordered by the group key and base key attributes of the framesets in the same order that they have been obtained. This will allow a sequential exploration of tuples retrieved to perform the reduction.

Example 4. Consider the example of an RVE depicted at the top of Figure 5 that contains a main query *Q1* (based on relation *S*) with a frame operator. *Q1* has two select predicates which involve subqueries *Q2* (based on *T*) and *Q3* (based on *U* with a frame operator). The corresponding reduction rule is depicted at the bottom of Figure 5.

The main query *Q1* forms groups: Transformation in Definition 4 is applied where *Q1* is the group relation, *g* is the group key, *S* is the source relation and *k1* is the source key. The frameset is composed by *g*, *k1*.

The resulting query still has two subqueries. *Q2* is a scalar subquery: Definition 7 is applied where base relation is *S*, source relation is *T* and base and source keys are *k1*, *k2* respectively, which form the second frameset.

Finally, the resulting query has a single subquery *Q3* which forms groups. According to Section 4.3 (*Case 4*), Definition 4 is first applied where *Q3* is the group relation, *h* is the group key, *U* is the source relation and *k3* the source key. Definition 7 is applied where S is the base relation, *U* is the source relation and *k1*, *k3* are the base and source keys, respectively. A correlation frame predicate *b<sum(d)* is created associated to this group. The third frameset is composed by *h*, *k1*, *k3*, *b*, *d*. Note that *b* and *d* have been added as they are needed to evaluate the frame predicate.

Reduction procedure. After obtaining the result set that contains tuples retrieved by the execution of the reduction rule, the problem is to detect and explore every frame from the initial database to select subsets of tuples based on the fitness functions and the costs. We can view these tuples as a logical table that is called *reduction tableaux*. Figure 6 displays a reduction tableaux for the previous example: Columns represent the framesets and their attributes (greyed columns include other attributes of the relations, although they do not belong to the framesets). Rows (tuples) and columns are partitioned into frames. From the figure we can see that each frame partitions the tuples that belong to its left frame based on the repeated values for keys of the frameset (repeating values of tuples inside a frame are not shown at each row for clarity).

Reduction is performed by exploring the reduction tableaux, taking into account:

- Tuples are processed sequentially: only a current tuple needs to be fetched from the database at each time.
- Each frameset maintains two sets of frames to keep track of the best and current solutions, respectively. This is the only data structure that is kept in memory
- We say that a frame or a frameset is at the end/beginning when the tuple below/above the current tuple belongs to a different frame, respectively.

To find a solution (tuples to be inserted in the reduced database) the evaluation proceeds over each frameset in the tableaux from left to right:

- When a frame begins to be evaluated the evaluation of its right frame is triggered before processing any of its tuples.
- When a frame ends its evaluation the current solution is added or replaces the current solution of this frame based on the cost.
- After the end of the evaluation of a frame, if its left frame is not at the end, the evaluation of the next frame begins again. Otherwise, the end of evaluation of the left frame is triggered.

A solution composed by the best solution of each frameset is found after the processing of the first frameset ends. Then, if there are more tuples the processing begins again to search for better solutions based on the cost.

Example 4 (cont.). Let us illustrate the above process using the reduction tableaux displayed in Figure 6. We consider in this example that the reduced database contains one tuple of *T* with *k2*=2 from the reduction of a previous rule. The following paragraphs detail the reduction process.

Tuple Id	Frameset 1(Q1)				Frameset 2 (Q2)			Frameset 3 (Q3)				
	g	k1	a	b	k1	k2	c	h	k1	k3	b	d
1	1	1	4	6	1	1	4	1	1	1	6	2
2										2		2
3										3		2
4										4		2
5					1	2	4	1	1	1	6	2
6										2		2
7										3		2
8										4		2
9	2	2	5	3	2	3	5	1	2	1	3	2
10										2		2
11										3		2
12										4		2

Figure 6. Example of a reduction tableaux for nested RVEs

The reduction starts at tuple 1 and frameset 1 which triggers the evaluation of framesets 2 and 3. Frameset 3 begins processing its tuples. Tuple 1 is included in its current solution for frameset 3, although the frame predicate is not covered yet. The process continues by evaluating frameset 3 with tuple 2. The correlation frame predicate $b<sum(d)$ improves the fitness and then tuple 2 is added to its current solution. The process continues by adding tuple 3 and finally tuple 4. At this point the frame predicate is true and there are no more tuples. A solution for frameset 3 has been found. Cost is evaluated as the sum of tuples to be inserted from the base tables: one for frameset 1, one for frameset 2 and 4 for frameset 3: $cost$=1+1+4=6. At this point, frameset 2 is at the end and then its current solution is stored as the best solution (same solution).

Frameset 1 is not at the end, therefore the evaluation of frameset 2 (2nd frame) is triggered, which triggers again the evaluation of frameset 3. Tuples 5, 6, 7 and 8 are processed as before until the solution is found. At this point frameset 2 is at the end, but now the cost is 1+0+4=5 (as the tuple with $k2$=2 in T was previously present in the reduced database). Then the best solution for frameset 2 is replaced by this one. Frameset 1 is at the end, so, its solution is also recorded as the best solution.

Row 9 restarts the evaluation of all framesets as in previous paragraphs, but in this case only two tuples are needed to make true the correlation predicate $b<sum(d)$ in frameset 3. The cost of this solution is 1+1+2=4. As at this point frameset 2 is at the end, this solution is also stored as its best solution. Likewise, frameset 1 is at the end, and the same solution replaces the previous best solution.

The final solution is the best solution for framesets 1 to 3, which include tuples 9 and 10. Looking at the primary keys in the tableaux we determine the tuples which have to be inserted in the reduced database: $k1$=2 (relation S), $k2$=3 (relation T) and $k3$=1, $k3$=2 (relation U).

Handling Views. The above reduction transformations may handle base relations (i.e. those which are implemented as tables) or derived relations (i.e. those which are composed by the result of evaluating a query). However, in commercial DBMS systems, named derived relations (called views) are frequently used. The procedures described before are applicable in this case, but a preprocessing stage is needed before transforming the coverage rules into the reduction rules. Whenever a view is found in a coverage rule, 1) the RVE that describes it is extracted, 2) transformed as shown above, 3) an auxiliary view with the transformed RVE is created with a different name (reduction view) and 4) the coverage rule is modified by replacing the name of the original view by the name of the reduction view.

5 Optimization and Tool Support

The reduction transformations and procedures shown in previous sections have been implemented in the QAShrink[1] tool, which automates the whole reduction process, including the generation of coverage rules, transformation into reduction rules and execution of the reduction procedures. Once the reduction procedures for all rules have finished, QAShrink selects the rows (tuples) from the initial database that have to be inserted in the reduced database and determines what other rows have to be added in order to preserve the integrity constraints. Finally, all these selected rows are copied from the initial database to the reduced database. The script to do this can be saved to allow the performance of further resets of the reduced database.

The reduction rules are SQL queries that are executed over the initial database to produce result sets that are processed by the reduction procedures. There are a number of factors affecting time performance of the reduction: 1) The query execution time especially for complex queries and when many rows have to be processed. 2) The row fetch time as most of the time spent in the transport of data between the database management server and the client machine is idle time waiting for the transference of a new data block. 3) The time spent in executing the reduction procedures. The rest of this section deals with the optimizations that have been implemented in QAShrink to improve its efficiency.

Three main kinds of optimizations may be made, which are detailed in subsequent sections:

1. Manipulate the reduction rule in order to achieve a faster execution in the database server. This will be accomplished by making some transformations on the reduction rule (Section 5.1).
2. Manipulate the reduction rule to limit the size of the reduction relation, i.e. the number of tuples returned by the reduction rule (Section 5.2). The use of this optimization can be optionally selected by the user.
3. Parallelize some operations in order to take advantage of multicore architectures and the load distribution between client and database server machines, i.e. the execution of the reduction procedures and the reduction rules, respectively (Section 5.3). The use of this optimization can be optionally selected by the user.

5.1 General efficiency optimizations

Coverage rules for a query are designed to obtain a subset of rows that satisfy a given test requirement for a query. This produces fewer rows than the original query and as such a faster processing. However, the transformations that obtain the reduction rules introduce an overhead due to the additional joins used to determine the base keys. A number of optimizations are made after the reduction rules have been generated:

[1] QAShrink is available at: http://in2test.lsi.uniovi.es/sqltools/qashrink/

Frame removal: If a rule does not have any select after a frame, the frames are removed as the reduction procedure only needs a single tuple from the frame to cover the rule.

Using SQL windowing functions[2] *for frames*: It is applicable only if the DBMS supports such functions. When this optimization is applied the clause PARTITION BY … OVER is used instead of joining the source and group relations.

Move subqueries to join clauses: If an uncorrelated scalar subquery appears at a conjunctive selection predicate, the condition containing the subquery is used for joining the source and base relations. This helps the DBMS to retrieve the data faster.

Convert non-scalar to scalar subqueries: In the above case, if the aggregate function is *avg*, *max* or *min* and the relational operator is < or ≤, an additional condition is added to the join predicate to force the subquery to return to the maximum value (conversely if operator is > or ≥). This decreases the number of tuples that are retrieved from the database and additionally avoids potential coverage losses.

Simplification of subquery rules: Some coverage rules for uncorrelated subqueries include a main query and a where clause with the subquery inside of an *exists* logical predicate. As at least one row that satisfies the main query has been obtained when processing the previous rules, the current rule is simplified by removing the main query.

5.2 Limiting the size of the reduction relation

A smaller size of the reduction relation is achieved by modifying the reduction rule to specify a limit in the number of tuples of different frames. Four different cases can be specified for:

1. *Queries without frames*: To limit the tuples retrieved by the reduction rule described in Section 4.1.
2. *Queries with frames*. To limit the tuples retrieved by the group relation described in Section 4.2.
3. *Frames*. To limit the tuples retrieved by the source relation described in Section 4.2.
4. *Subqueries*: To limit the tuples retrieved by the source relation described in Section 4.3.

This is accomplished by enclosing the SQL of the reduction rule under the clause PARTITION BY … OVER. This allows controlling the size of each frame by specifying its maximum number of tuples. Note that this optimization can only be used if it is supported by the DBMS. The particular syntax depends on the particular DBMS vendor specification. For instance, in SQL server the TOP keyword is used after SELECT. In Oracle, the ROWNUM special column is added in a condition of the WHERE clause.

Limiting the result size is a trade-off between cost and quality. When limiting the result set size the efficiency improves as less data is read from the dataset. However, this implies that less data may be reused when performing the reductions and so, a larger reduced database may be produced.

5.3 Parallelizing tasks

Consider a database reduction that has to process a number of reduction rules δ_i. The client computer applies the reduction transformations to generate a rule δ_1 that is sent to the database server for execution. It receives the result from the database server and then executes the reduction procedure. After finishing, the next rule δ_2 is sent to the database and so on. Although some processes may be executed in parallel (database server may parallelize some tasks and rows can be sent to the client while it is still processing a query) the smallest time is constrained by the sum of the execution times of each rule δ_i.

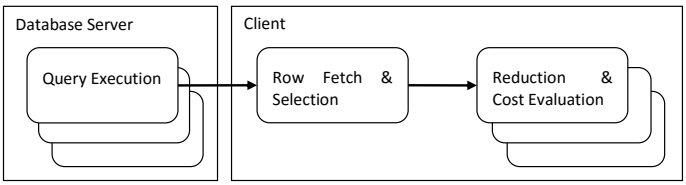

Figure 7. Parallelizing the reduction procedures

To allow a faster processing we should parallelize some other tasks. For example, assume that we send four rules to be executed at the database server in parallel. At this moment, the client is idle waiting for some result set. At some time, two rules begin returning result sets to the client. Then, two reduction procedures are now being executed in parallel. As soon as one of these procedures finishes (e.g. δ_1), a new rule (δ_5) is issued to the database server and so on. Now the total time is not constrained by the sum of execution times of all rules. This design is detailed below. Figure 7 depicts the different tasks that may be executed as different subprocesses (threads). These optimizations are not dependent on the DBMS and therefore they can be applicable in all cases.

Parallelizing Query Execution: Several queries (reduction rules) are submitted in parallel for execution to the database server. Having several processes in parallel allows decreasing the total query execution time in the database server and an additional decreasing of the time that the reduction procedure must wait between issuing a query for execution and the fetching of the first result set row.

Parallelizing Reduction & Cost Evaluation: Each thread executes the reduction procedure for a rule, evaluates the cost and decides whether a row will be kept to be inserted in the reduced database. This decreases the total time of the reduction procedures.

Row Fetch & Selection: This process coordinates the overall reduction process. It receives each open result set from the execution of reduction rules and sends rows to the Reduction & Cost Evaluation threads. As soon as it receives the latest row from a rule, it issues the next rule for reduction. This does not preserve the order in which rules are reduced, but contributes to keep all processors as busy as possible.

6 Evaluation

To assess the feasibility of the approach to database reduction we present the results of the reduction obtained with three different databases as case studies.

6.1 Research Questions

This evaluation addresses four primary exploratory questions related to the effectiveness:

- RQ1: How does the reduction perform in terms of fault-

[2] The windowing functions (eg. PARTITION BY) were introduced by the ANSI/ISO SQL:2003 standard. They are supported by DBMSs such as Oracle, SQLServer or PostgreSQL.

TABLE 1. DETAILS OF THE DATABASES USED IN THE STUDY

Database name	DBMS	Tables	Rows (all tables)	Rows (largest table)	Queries	Views
Helpdesk	SQL Server	31	137,490	103,553	198	7
Compiere	Oracle	129	127,200	1,000	107	5
TPC-H (5 instances)	Oracle	8	86,805 to 866E+6	6,175 to 600E+6	22	1

TABLE 2. DETAILS OF SOME QUERIES USED IN THE STUDY

Database name	Query name/id	Tables	G	U	V	S	C	W
Helpdesk	17	10						2
	18	11			1		2	11
	29	8		1		2		4
	50	2				1		4
Compiere	RV_BPartnerOpen	7		2	2		14	4
	C_Invoice_Candidate_v	5	1			1		24
	C_Invoice_LineTax_vt	24		5			13	5
	RV_UnPosted	16		15				15
TPCH	7	6	1			1		11
	8	8	1			1	1	10
	20	5				3		10
	21	6	1			2		13

detection effectiveness at the query level?

- RQ2: How does the reduction perform in terms of coverage at the query level?
- RQ3: How does the reduction perform in terms of reduction effectiveness?
- RQ4: How does the reduction perform at the application level?

Two additional secondary questions related to the efficiency that may have influence on RQ1 to RQ4 are addressed:

- RQ5: Is the approach scalable with the size of the database?
- RQ6: How do the optimizations affect the performance of the reduction?

6.2 Objects of Study

This evaluation uses three different initial databases (Helpdesk, Compiere and several instances of TPC-H from 10MB to 100GB) and their associated queries. Table 1 displays for each database the DBMS used, the number of tables, its size in terms of the total number of rows for all tables as well as the size of the largest table. The last columns show the number of queries that will be used in this study and the number of views that are used.

Table 2 displays more detailed metrics for a number of representative queries, including the total number of tables involved in the query, the number of groupings (G), union statements (U), views (V) and subqueries/derived tables (S). The complexity of decisions is measured by counting the number of case statements (C) and the number of conditions in where (W).

Helpdesk is an in-house web application to manage service requests [7]. We used a production database as the initial database with 22,387 tickets, 103,553 annotations on tickets and 279 users. The SQLServer database contains 31 tables. The set of queries has been taken from the database logs collected during security testing sessions, comprising 198 different queries.

Compiere[3] is an open source ERP and Customer Relationship Management (CRM) business solution for Small and Medium-sized Enterprises. We used as initial database a randomly generated Oracle database containing 129 tables. The tables have an average of 23 columns, the largest having 84 columns. The set of queries is composed of the full set of views of the application (107 queries).

TPC-H [97] is a benchmark to evaluate the performance of the execution of queries against databases. It contains 8 tables and 22 queries[4] which cover most SQL constructs. Queries are designed to stress the DBMS and include different combinations of joins, groupings, different kinds of subqueries and derived tables. We divided the queries into two objects of study (11 queries each): TPCHg which mainly contains groups without subqueries and TPCHs which contains subqueries, groups and a view.

This study has been performed using two 4 core Intel Xeon X5660 2.8GHz virtual machines. The QAShrink application uses a 3GB virtual machine. The database server uses SQL Server 2008 R2 and Oracle Database 11g Enterprise V11.2.0.1.0 with 6GB reserved for the database. In order to avoid the bias caused by database cache between experiments, every run has been preceded by a complete shutdown of the database server.

6.3 Effectiveness at the query level (RQ1 to RQ3)

Table 3 displays the main results related to RQ1 and RQ2. To assess the fault-detection effectiveness we performed a mutation analysis. Previous studies showed that mutation analysis is an appropriate method for evaluating the fault detection capabilities of a test suite [98][99]. As we deal with SQL queries we generated a set of mutants for each query using the SQLMutation tool [22]. It applies two main kinds of query mutation operators (described in detail in [17]):

- Conventional mutations on relational, logical and arithmetic operators in conditions and expressions, and replacement of identifiers.
- SQL specific mutations on main SQL clauses (joins, subqueries, aggregates, etc.) and null values.

Both mutation score and SQLFpc coverage were measured against the initial and the reduced databases.

RQ1 (fault detection effectiveness). From Table 3 we can appreciate that the reduction produces a decrease in mutation score, small in some databases and larger in others (maximum value is 6.6%). These results show fairly similar effectiveness losses to other results for non-database applications described in Section 2.1 which show up to 7.3% effectiveness losses. That means that the reduced database contains a diverse set of rows that may be considered good enough to be used for testing purposes in the sense that they have similar fault detection ability measured in terms of mutants.

RQ2 (coverage). At first glance, when considering the SQLFpc coverage the expected result would be to achieve a lower or equal coverage than using the initial database, but the actual result shows that coverage increases in all reduced databases (maximum value 4.9%). This increasing of coverage depends on

[3] The source distribution of Compiere can be found at http://sourceforge.net/projects/compiere. The set of views used in this study can be found in the file compiere-all\db\database\Create\Views.sql of the Version 2.53b.

[4] The specification of the the database schema and all queries for the latest versión of TPC-H can be found at http://www.tpc.org/information/current_specifications.asp under sections 1.2 and 2.4, respectively.

TABLE 3. COVERAGE AND MUTATION SCORE BEFORE AND AFTER

	Compiere	Helpdesk	TPCHg (10MB)	TPCHs (10MB)
Number of mutants	192,851	65,666	4,835	3,069
Mutation Score (initial)	86.3%	61.8%	81.2%	76.6%
Mutation Score (reduced)	79.7%	61.5%	78.9%	70.0%
Mutation Score difference	-6.6%	-0.3%	-2.3	-6.6%
Number of rules	1,762	1,341	145	119
Coverage (initial)	58.3%	49.0%	63.4%	68.9%
Coverage (reduced)	58.6%	50.5%	68.3%	71.4%
Coverage difference	+0.3%	+1.5%	+4.9%	+2.5%
Coverage Gain	6	20	7	4
Coverage Loss	2	0	0	1

TABLE 4. SIZE OF INITIAL AND REDUCED DATABASES

	Compiere	Helpdesk	TPCHg (10MB)	TPCHs (10MB)
Size (initial)	127,200	137,490	86,805	86,805
Size (reduced)	1,356	194	220	119
Reduction Factor	1.07%	0.14%	0.25%	0.14%
Rows per Covered Rule	1.32	0.29	2.39	1.45

the coverage gains (a coverage rule is not covered when executed against the initial database, but covered when executed against the reduced database) and coverage losses (a coverage rule is covered when executed against the initial database, but uncovered when executed against the reduced database). The last rows in Table 3 display each of these values (number of rules that become covered or uncovered, respectively, after the reduction).

Coverage gains are due to data not selected from the initial database in detail tables as explained in Section 4.1 and coverage losses are caused by aggregate functions as explained in Section 4.2. From the data in Table 3 we can conclude that for the databases used in this study the reduced database is a reasonably good solution in terms of coverage, although with a few coverage loss for some rules.

RQ3 (reduced database size). Table 4 displays the size of all databases (both initial and reduced). The size of the databases is measured as the sum of the number of rows for all tables. We can appreciate a significant reduction factor that leads to very slight reduced databases.

As the reduction is driven by the coverage rules, which in turn depend on the number of queries and their size and complexity, the absolute size of the reduced database is not the best indicator for the effectiveness of the reduction. The last row in Table 4 displays the average number of rows taken from the initial database that have to be inserted in the reduced database divided by the total number of covered rules. We can observe that these figures are small, ranging from 0.29 for the simplest queries (Helpdesk) to 2.39 for the more complex queries (TPCHg).

6.4 Effectiveness at the application level (RQ4)

The previous section showed the effectiveness related to coverage and mutation score for the queries involved in the reduction. If part of the application logic is embedded in queries, the output of a single run of a test case will depend on the values returned by the queries and on the decisions taken in the procedural code.

TABLE 5. MUTATION COVERAGE (APPLICATION LEVEL)

	Initial database	Reduced database	Reduced database (2nd)
Size (number of rows)	137,490	194	65
Num. of ticket records	22,387	31	10
Decision coverage	100%	100%	100%
Num. of mutants (MAJOR)	70	70	70
Mutation score (MAJOR)	58.6%	58.6%	58.6%
Num. of mutants (muJava)	320	320	320
Mutation score (muJava)	84.4%	85.6%	84.2%

RQ4 (Effectiveness at the application level). To check this question we prepared a set of test cases for the security check functionality of the Helpdesk application. Security is the most critical functionality as the ability to access tickets and annotations depends on many factors:

- The logical database that the user has permission to access, its organizational unit, the allowed ticket types and the previous annotations made on tickets.
- Other user level parameters: restricting access to only own tickets or only tickets that belong to its organizational unit.
- The transaction type (read, update, insert).

The security checking has a single method as entry point plus 3 auxiliar methods coded in Java containing 15 decisions. During the execution, the queries are dynamically constructed at 9 places in the code in order to take the appropriate decisions based on the outputs produced by these queries.

We designed and automated (with Junit) a set of test cases. The design of the test cases was made using a black-box approach taking into account the Helpdesk security requierements. From these security requirements we derived the test requirements and then the test cases until all test requirements were met. As a result, we obtained a total of 31 test cases. Instead of creating a new test database for developing the test cases, we used the existing reduced Helpdesk database (Table 4). This task is fairly straightforward as the reduced database is small enough to manually find the records that represent each test requirement by browsing the existing ones.

Next, the set of queries that were issued to the database when executing these test cases were used to perform a second reduction process of the initial database using these queries, leading to a second reduced database with 65 rows. The first two rows in Table 5 display the sizes of the databases as well as the number of ticket records contained in each database.

The execution of the test cases against each of the test databases leads to the same decision coverage. To evaluate the application level effectiveness we performed a mutation analysis on the four Java methods under test using two different tools: MAJOR V1.1.6 [100][101] and muJava V4 [102]. Using each database as the test database, the test cases were executed against each mutated version. The results are displayed in the last rows in Table 5.

MAJOR generated less mutants that led to lower scores than muJava, as it is designed to maximize the efficiency of mutation analysis [103][104][105]. The tests executed using the initial database achieve very similar or equal mutation score than the tests executed against reduced databases, regardless of the tool used to generate the mutants (58.6% using MAJOR and around 85% using muJava). Moreover, with muJava the mutation score is slightly lower than 85% when using different databases than

TABLE 6. SQLFPC COVERAGE GAINS - LOSSES

	Compiere	Helpdesk	TPCHg 10MB	TPCHg 100MB	TPCHg 1GB	TPCHg 10GB	TPCHg 100Gb	TPCHs 10MB	TPCHs 100MB	TPCHs 1GB	TPCHs 10GB	TPCHs 100GB
NoLim	6-2	20-0	7-0	9-0	7-0	4-0	6-0	4-1	4-2	4-3	6-1	4-1
Lim/1000	6-2	19-0	5-0	9-0	7-0	4-0	6-0	4-1	4-2	4-3	6-1	5-1
Lim/100	6-3	20-0	5-0	9-0	5-0	4-0	4-3	4-1	4-2	4-3	6-2	5-2
Lim/10	6-4	20-0	4-0	9-0	7-1	3-3	4-5	4-2	3-2	3-3	8-5	5-5

the one used to build the test cases (the reduced database). The reason is that some mutants result in modifications on part of the SQL generated by the program (e.g. by removing a clause). On a few occasions these mutants remain live when executed against different databases. Nevertheless, the score for mutants that modify other parts of the application logic is the same across all test databases.

6.5 Efficiency (RQ5 and RQ6)

RQ5 (Scalability). To check the scalability of the reduction with respect to the size of the database we generated a set of TPCH databases using several scale factors, leading to 5 Oracle databases ranging from 10 MB to 100GB (each database multiplies the size of the previous by 10). This leads to databases ranging from 86,805 to 865,860,820 rows. We also repeated several runs using all the optimizations that limit the size of the reduction relation (Section 5.2) by limiting the number of tuples to 10, 100 and 1,000, as well as not using any limit (NoLim).

Figure 8 displays the coverage of the initial and reduced databases. In general, the coverage is slightly larger for the reduced databases than for the initial database. The increase is smaller as the limit of the size of the reduction relation is also smaller, due to the fact that fewer rows that have been covered by previous rules are reused.

Table 6 displays the details of coverage gains and losses. For Compiere and Helpdesk the gains and losses do not change significantly with different optimization parameters. For TPCHg and TPCHs there are differences for different scaling and optimization parameters: Coverage gains are in a range from 3 to 9 without a defined pattern. The coverage losses range from 0 to 5 with a strong dependency on the optimization parameters. Consider the worst case (database with 100GB and limiting the frame to 10 rows). Using the initial database some frames are composed by many thousands of rows. Selecting only ten of them increases the probability of not fulfilling the select predicates after the frames, causing the coverage losses. There is a compromise on the loss of effectiveness caused by these kinds of optimization and the efficiency improvement that will be shown later. However, in most cases losses are small and in this worst case losses compensate the gains.

Figure 9 depicts the size of each database after the reduction (measured as the total of rows in all tables). The size slightly increases with the optimizations and it is fairly independent from the size of the database for the largest ones (TPCH with 1GB to 100GB). This leads to a high efficiency of the reduction with little dependency on the size of the initial databases and the optimizations, showing very large reductions measured in the percentage of rows that are kept in the reduced database (up to 0.000035% for TPCHg and 0.000017% TPCHs).

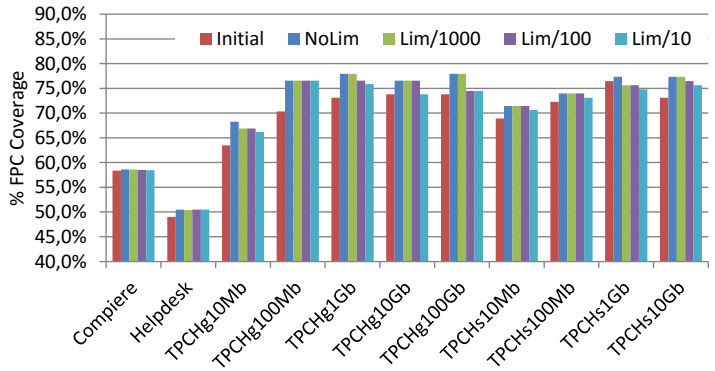

Figure 8. SQLFpc Coverage of each database after reduction

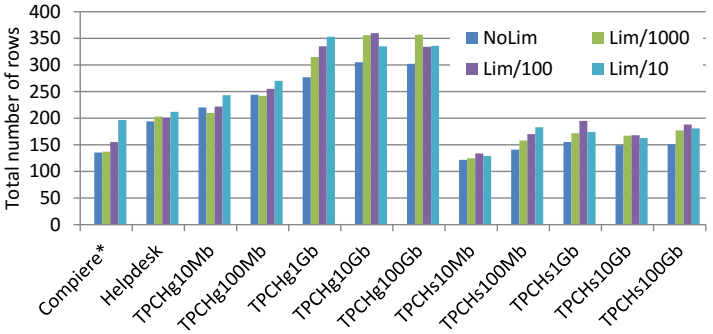

Figure 9. Size of each database after reduction (number of rows inserted in the reduced database). *The number of rows for Compiere has been divided by ten.

RQ6 (Performance). To check the performance with respect to the size of the database and the optimizations, we repeated several runs using the optimizations presented in Section 5.3 (Seq means sequential and Par means parallel) as well as the row limits used in the previous section.

Figure 10 depicts the total time spent in the reduction in log scale. The figure shows how the time performance scales with the database size for TPCH. The relation of time spent when reducing each database with respect to the previous one (considering data for Seq) is 4.8, 10.8, 10.9, 13.4 for TPCHg and 5.0, 11.8, 11.5, 12.3 for TPCHs. As each database multiplies the size of the previous one by a factor of ten we can see a near-linear growth of the time with respect to the size.

A reduction rule is generated by transforming a coverage rule which in turn is generated by transforming the original query. In order to compare times across different databases and different sets of queries we normalize the execution times with respect to the original query. Figure 11 depicts the average time spent per reduction rule divided by the average time spent per original query.

Normalized times are between 1 and 3.5 across all databases (Seq). Considering the results for TPCH, the normalized execution times are very similar, but this value suddenly rises for the largest database (100GB) and this effect occurs early in TPCHs (10GB) which contain subqueries. This is caused by the additional joins needed to reveal the base keys as well as the amount of RAM memory available to the database server. Up to TPCH 1GB data needed for processing the query has enough room to be kept in RAM and then the reduced queries run faster. For larger database sizes the memory may not be sufficient and secondary storage (temporary space) has to be used, penalizing the

performance. The effect of optimizations that limit the number of rows strongly decreases these times, the decrease being higher for the largest databases, keeping the normalized execution time near or below 1. This effect is similar for Compiere and Helpdesk databases, but not so pronounced.

Parallelizing also has a strong influence on the performance by reducing roughly by half (on most occasions) the execution times. The benefits of optimizations are larger for the biggest databases. The greatest improvement is for TPCHs 100GB which has a normalized execution time of 3.43 (sequential without row limits) that decreases to 0.40 (parallel with row limit 10) which means that overall reduction time decreases by 11.7%.

7 Threats to Validity

Results from the above study show a high degree of reduction of the databases while preserving most of the coverage with a suitable performance and scalability. Transformation rules cover many of the typical SQL constructs and their combinations, and are fully automated. The criterion is white-box based, therefore, the typical usage scenario of the approach is mainly driven towards maintenance or reengineering as it requires a set of queries and enough data to be loaded in the database to be reduced. However, there are several issues that may threaten the validity of these results, which are discussed below.

First, the reduction is driven by the SQLFpc coverage. There is no guarantee that the test requirements embodied in the criterion are the best suited for testing SQL queries, but as it is based on the principles of MCDC it is more likely that they are reasonably adequate for determining a number of interesting situations to test.

Second, the coverage of the reduced database is not always the same as the initial database, although the variations shown before are small. On some occasions the coverage increases (coverage gain). On others there is a possibility that rules covered against the initial database, become uncovered over the reduced database (coverage loss). The frame reduction procedures use an approach to reduce frames which is not intended to find an optimum, but a small enough frame, and under some circumstances it may fail to find a solution. In this case, the bigger the database is (containing more frames), the more likely it is to find at least a small enough reduced frame. As shown before, the effect of coverage losses is present, however, it has been observed for a small percentage of coverage rules as most of them are additive, i.e., adding rows to cover a rule does not prevent other rules from being covered.

Third, the approach of the reduction is based on the coverage at the query level. This is well suited for reporting when most of the application logic resides in the queries, but in general does not guarantee the coverage of the procedural code of an application. The more application logic is embedded in the queries and the simpler decisions based on the queries are, the more likely it is that coverage of the procedural code be kept because the reduced database has considered the coverage of the logic embedded in the query. This has been considered in the evaluation of the effectiveness at the application level, although the manual design of the test cases as well as the kind of application used, is another threat to validity. Another potential problem is that coverage of the queries may be modified by previous changes to the data made by the program. It is important to note

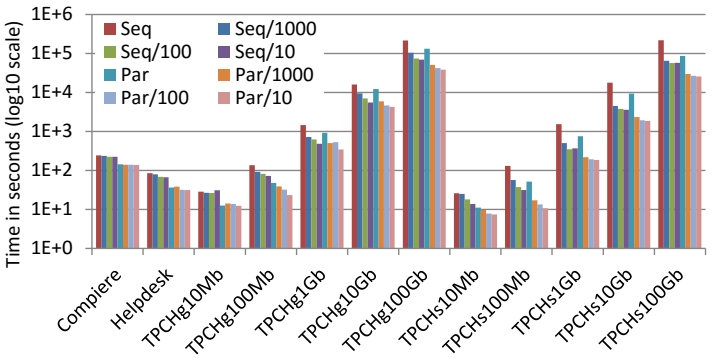

Figure 10. Total time (seconds) spent in the database reduction (depicted in Log scale)

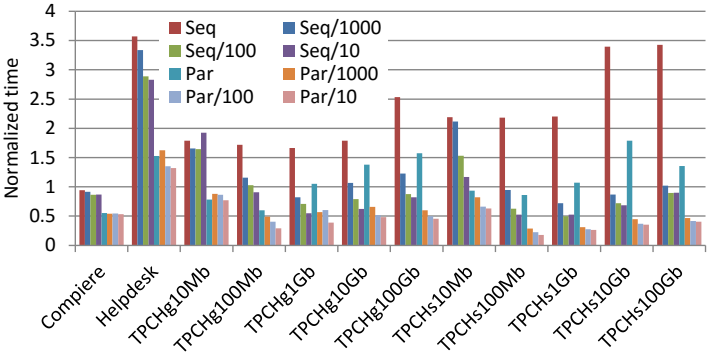

Figure 11. Normalized execution time spent in the database reduction (average reduction time per rule divided by the average execution time per query)

that the reduced database is intended to be a starting point for testing, but the tester may need to insert additional data for testing particular situations.

Fourth, although many combinations of SQL clauses have been considered, the total number of combinations is potentially infinite meaning that a reduction transformation may fail for some rules, leading to uncovered rules. To mitigate this effect the automation has been thoroughly tested.

Finally, the study is limited to three applications, so the results may not generalize to other applications. However, these are real-life applications that use SQL to process some logically complex queries (in Helpdesk, related to the access security and in Compiere ERP, the set of views on the basis of which all other queries of the application are constructed). Additionally the TPCH is an industry benchmark for testing the performance of complex queries.

8 Conclusion

We have presented an approach for the reduction of test databases that takes a set of SQL queries and an initial database, and produces a reduced test database that preserves the SQLFpc coverage. The approach is able to handle complex queries covering a large set of SQL constructs and their combinations.

The results showed that a high degree of reduction can be attained with few coverage losses and some coverage gains. Ad-

ditionally, it is scalable in relation to the size of the initial database and the reduction time. Moreover, the whole approach to the reduction of a database is fully automated and some optimizations are included.

The typical target scenario is composed by applications that rely on SQL queries for processing complex business rules. A reduced test database may be created for testing or developing specific queries, leading to a starting point of the test database to complete the tests. When considering the whole application, a large set of queries may be extracted from the database logs in order to create a reduced database covering these queries, which in turn may be used as a starting point for completing tests or performing other maintenance operations. This is the first potential benefit of the approach that contributes to a reduction of the time spent on the task of creating test databases. A second potential benefit is to allow a decrease of the times of loading test databases, while keeping a representative set of data to exercise the queries of the applications, leading to faster test execution. Additionally, having small test databases contributes to make the task of checking the actual results easier when developing and testing queries, contributing to a faster and more reliable test results comparison.

Our future work will concentrate on two areas. First, to complete the evaluation and practical use at the application level to include the ability to reduce test databases into the developer and tester workflows. This will imply testing how the reduction performs using other DBMS. Second, to use the reduction principles to address NoSQL databases in order to provide support for testing in the context of the development of applications that manipulate data using these technologies.

ACKNOWLEDGMENT

This work was supported in part by projects TIN2010-20057-C03-01 and TIN2013-46928-C3-1-R, funded by the Spanish Ministry of Economy and Competitiveness, and GRUPIN14-007, funded by the Principality of Asturias (Spain) and ERDF funds.

REFERENCES

[1] A.B.M. Moniruzzaman and S.A. Hossain, "NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison", *International Journal of Database Theory and Application*, vol. 6, no. 4, Aug. 2013.

[2] *ISO/IEC 9075, Information technology - Database languages - SQL*. International Standards Organisation, 1999.

[3] *ISO/IEC/IEEE 29119-1:2013 Software and systems engineering - Software testing - Part 1: Concepts and definitions*. International Standards Organisation, 2013.

[4] J. Tuya, M. J. Suárez-Cabal and C. de la Riva, "Full predicate coverage for testing SQL database queries", *Software Testing, Verification and Reliability*. *vol*. 20, no. 3, pp. 237-288, Sep. 2010.

[5] RTCA Inc, *DO-178-B: Software Considerations in Airborne Systems and Equipment Certification*, Radio Technical Commission for Aeronautics (RTCA), 1992.

[6] J. J. Chilenski, *An investigation of three forms of the modified condition decision coverage (MCDC) criterion*, Technical Report DOT/FAA/AR-01/18, U.S. Department of Transportation, Federal Aviation Administration, April 2001.

[7] J. Tuya, M. J. Suarez-Cabal and C. de la Riva, "Query-Aware Shrinking Test Databases", *Proc. Second Int'l Workshop on Testing Database Systems (DBTest'09)*, June 2009.

[8] S. Yo and M. Harman, "Regression testing minimization, selection and prioritization: a survey", *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67-120, Mar. 2012.

[9] G. Rothermel, M. J. Harrold, J. Ronne and C. Hong, "Empirical studies of test suite reduction", *Software Testing, Verification, and Reliability*, vol. 4, no. 2, pp. 219–249, Dec. 2002.

[10] E. E. Emelie, M. Skoglund M and P. Runeson. "Empirical evaluations of regression test selection techniques: A systematic review", *Proc. Second ACM-IEEE Int'l Symposium on Empirical Software Engineering and Measurement (ESEM'08)*, 2008.

[11] W. E. Wong, J. R. Horgan, S. London and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness", *Software Practice and Experience*, vol. 28, no. 4, pp. 347-369, Apr. 1998.

[12] W. E. Wong, J. R. Horgan, A. P. Mathur and A. Pasquini, "Test set size minimization and fault detection effectiveness: A case study in a space application", *The Journal of Systems and Software*, vol. 48, no. 2, pp. 79-89, Oct. 1999.

[13] G. Rothermel, M. J. Harrold, J. Ostrin and C. Hong, "An empirical study of the effects of minimization on the fault detection capabilities of test suites", *Proc. Int'l Conf. on Software Maintenance (ICSM'98)*, 1998.

[14] G. Rothermel, M. J. Harrold, J. Ronne and C. Hong, "Empirical studies of test suite reduction", *Software Testing, Verification, and Reliability*, vol. 4, no. 2, pp. 219-249, Dec. 2002.

[15] E. F. Codd, "A relational model of data for large shared data banks", *Communications of the ACM*, vol. 13, no. 6, pp. 377-387, Jun. 1970.

[16] E. F. Codd, *The Relational Model for Database Management - Version 2*, Addison-Wesley, 1990.

[17] J. Tuya, M. J. Suárez-Cabal and C. de la Riva C, "Mutating database queries", *Information and Software Technology*, vol. 49, no. 4, pp. 398-417. Apr. 2007.

[18] G. Kaminski, U. Praphamontripong, P. Ammann and J. Offutt, "A logic mutation approach to selective mutation for programs and queries", *Information and Software Technology*, vol. 53, no. 10, pp. 1137-1152, Oct. 2011.

[19] W. K. Chan, S. C. Cheung and T. H. Tse, "Fault-based testing of database application programs with conceptual data model", *Proc. 5th Int'l Conf. on Quality Software (QSIC'05)*, pp. 187-196, 2005.

[20] C. J. Wright, G. M. Kapfhammer and P. McMinn, "Efficient Mutation Analysis Of Relational Database Structure Using Mutant Schemata And Parallelisation", *Proc. 8th Int'l Workshop on Mutation Analysis*, 2013.

[21] C. J. Wright, G. M. Kapfhammer and P. McMinn, "The Impact of Equivalent, Redundant and Quasi Mutants on Database Schema Mutation Analysis", *Proc. 4th Int'l Conf. on Quality Software (QSIC'14)*, Oct. 2014.

[22] J. Tuya, M. J. Suárez-Cabal and C. de la Riva, "SQLMutation: a tool to generate mutants of SQL database queries". *Proc. Second Workshop on Mutation Analysis*, 2006.

[23] C. Zhou and P. G. Frankl, "Mutation testing for java database applications". *Proc. Second Int'l Conf. on Software Testing Verification and Validation (ICST'09)*, pp. 396-405, 2009.

[24] C. Zhou and P. G. Frankl, "JDAMA: Java database application mutation analyser", *Software Testing, Verification and Reliability*, vol. 21, no. 3, pp. 241-263, Sep. 2011.

[25] C. Zhou and P. G. Frankl, "Inferential Checking for Mutants Modifying Database States", *Proc. 4th Int'l Conf. on Software Testing Verification and Validation (ICST'11)*, pp. 259-268, 2011.

[26] S. R. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones and M. J. Harrold, "Localizing SQL Faults in Database Applications", *Proc. 6th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE'11)*, pp. 213-222, Nov. 2011.

[27] C. Zhou and P. G. Frankl, "Empirical Studies on Test Effectiveness for Database Applications", *Proc. 5th Int'l Conf. on Software Testing Verification and Validation (ICST'12)*, pp. 61-70, 2012.

[28] H. Shahriar and M. Zulkernine, "MUSIC: Mutation-based SQL injection vulnerability checking*", Proc. 8th Int'l Conf. on Quality Software (QSIC'08)*, pp. 77–86, 2008.

[29] P. Bisht, P. Madhusudan and V. N. Venkatakrishnan, "CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks". *ACM Transactions on Information and System Security*, vol. 13, no. 2, article 14, Feb. 2010.

[30] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for SQL injection vulnerabilities: an input mutation approach", *Proc. Int'l Symposium on Software Testing and Analysis (ISSTA'14)*, pp. 259-269, 2014.

[31] W. G. J. Halfond and A. Orso, "AMNESIA: analysis and monitoring for Neutralizing SQL-injection attacks", *Proc. 20th IEEE/ACM Int'l Conf. on Automated software engineering (ASE '05)*. 174-183, 2005.

[32] G. M. Kapfhammer, M. L. Soffa, "A family of test adequacy criteria for database-driven applications", *Proc. 9th European Software Engineering Conf. held jointly with 11th ACM SIGSOFT Int'l Symposium on the Foundations of Software Engineering*, pp. 98–107, 2003.

[33] D. Willmor and S. M. Embury, "Exploring test adequacy for database systems", *Proc. 3rd UK Software Testing Research Workshop*, pp. 123-133, 2003.

[34] G. M. Kapfhammer and M. L. Soffa, "Database-aware test coverage monitoring", *Proc. 1st India Software Engineering Conf. (ISEC'08)*, pp. 77-86, 2008.
[35] O. S. Leitao Jr., P. R. S. Vilela and M. Jino, "Data flow testing of SQL-based active database applications", *Proc. 3rd Int'l Conf. on Software Engineering Advances (ICSEA'08)*, pp. 230-236, 2008.
[36] M. J. Suárez-Cabal and J. Tuya, "Using an SQL coverage measurement for testing database applications". *Proc. 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'12)*, pp. 253-262, 2004.
[37] M. J. Suárez-Cabal and J. Tuya, "Structural coverage criteria for testing SQL queries", *Journal of Universal Computer Science*, vol. 15, no. 3, pp. 584-619, 2009.
[38] P. McMinn, C. J. Wright and G. M. Kapfhammer, "An Analysis of the Effectiveness of Different Coverage Criteria for Testing Relational Database Schema Integrity Constraints", *University of Sheffield, Dept. Computer Science*, http://www.dcs.shef.ac.uk/intranet/research/public/resmes/CS1501.pdf [3 Mar. 2015].
[39] W. G. J. Halfond and A. Orso, "Command-form coverage for testing database applications" *Proc. 21st IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE'06)*, pp. 69-80, 2006.
[40] G. Kaminski, G. Williams and P. Ammann, "Reconciling perspectives of software logic testing", *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 149-188, Jan. 2008.
[41] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.
[42] D. Chays, S. Dan, P. G. Frankl, F. U. Vokolos and E. J, Weyuker, "A framework for testing database applications". *Proc. ACM SIGSOFT Int'l Symposium on Software Testing and Analysis (ISSTA'00)*, pp. 147-157, 2000.
[43] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos and E. J Weyuker, "An AGENDA for testing relational database applications", *Software Testing, Verification and Reliability*, vol. 14, no. 1, pp. 17-44, Jan. 2004.
[44] Y. Deng, P. G. Frankl and D. Chays, "Testing database transactions with AGENDA". *Proc. 27th Int'l Conf. on Software Engineering (ICSE'05)*, pp. 78-87, 2005.
[45] D. Chays, J. Shahid and P. G. Frankl, "Query-based test generation for database applications", *Proc. First Int'l Workshop on Testing Database Systems (DBTest'08)*, 2008.
[46] D. Willmor and S. M. Embury, "An intensional approach to the specification of test cases for database applications", *Proc. 28th Int'l Conf. on Software Engineering (ICSE'06)*, pp. 102-111, 2006.
[47] D. Willmor and S. M. Embury, "Testing the implementation of business rules using intensional database tests" *Proc. Testing: Academic & Industrial Conf. on Practice and Research Techniques (TAIC PART'06)*, pp. 115-126, 2006.
[48] C. Binnig, D. Kossmann and E. Lo, "Reverse query processing", *Proc. 23rd Int'l Conf. on Data Engineering (ICDE'07)*, pp. 506-515, 2007.
[49] C. Binnig, D. Kossmann and E. Lo, "MultiRQP - Generating test databases for the functional testing of OLTP applications" Proc. 1st Int'l Workshop on Testing Database Systems (DBTest'08), 2008.
[50] W. T. Tsai, D. Volovik D and T. F. Keefe TF, "Automated test case generation for programs specified by relational algebra queries", *IEEE Transactions on Software Engineering*, vol. 16, no. 3, pp. 316-324. Mar. 1990.
[51] J. Zhang, C. Xu and S. C. Cheung, "Automatic generation of database instances for white-box testing", *Proc. 25th Int'l Computer Software and Applications Conf. (COMPSAC'01)*, pp. 161-165, 2001.
[52] S. A. Khalek, B. Elkarablieh, Y. O. Laleye and A. Khurshid, "Query-aware test Generation using a relational constraint solver", *Proc. 23rd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE'08)*, pp. 238-247, 2008.
[53] S. Shah, S. Sudarshan, S. Kajbaje, S. Patidar, B. P. Gupta and D. Vira, "Generating test data for killing SQL mutants: A constraint-based approach", *Proc-27th IEEE Int'l Conf. on Data Engineering (ICDE'11)*, pp. 1175–1186, 2011.
[54] P. Vemasani, A. Brodsky and P. Ammann, "Generating Test Data to Distinguish Conjunctive Queries with Equalities", *Proc. of IEEE Software Testing, Verification and Validation Workshops*, pp. 216-221, 2014.
[55] C. de la Riva, M. J. Suárez-Cabal and J. Tuya, "Constraint-based Test Database Generation for SQL Queries". *Proc. 5th Int'l Workshop on Automation of Software Test (AST'10)*, 2010.
[56] M. Emmi, R. Majumdar and K. Sen, "Dynamic Test input generation of database applications", *Proc. Int'l Symposium on Software Testing and Analysis (ISSTA'07)*, pp. 151-162, 2007.
[57] K. Pan, X. Wu and T. Xie, "Program-input generation for testing database applications using existing database states". *Automated Software Engineering*, pp. 1-35, Jul. 2014.
[58] K. Pan, X. Wu and T. Xie, "Generating program inputs for database application testing", *Proc. of IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE'11)*, pp. 73–82, 2011.
[59] K. Pan, X. Wu and T. Xie, "Guided test generation for data-base applications via synthesized database interactions", *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 2, article 12, Mar. 2014.
[60] C. Li, and C. Csallner, "Dynamic symbolic database application testing", *Proc. of Int'l Workshop on Testing Database Systems (DBTest'10)*, pp. 1-10, 2010.
[61] G. M. Kapfhammer, P. McMinn and C. J. Wright, "Search-Based Testing of Relational Schema Integrity Constraints Across Multiple Database Management Systems", *Proc. IEEE 6th Int'l Conf. on Software Testing, Verification and Validation (ICST'13)*, pp. 31-40, Mar. 2013.
[62] N. Bruno and R. V. Neheme, "Finding Min-Repros in Database Software", *Proc. Second Int'l Workshop on Testing Database Systems (DBTest'09)*, June 2009.
[63] K. Morton and N. Bruno, "FlexMin: A Flexible Tool for Automatic Bug Isolation in DBMS Software", *Proc. Fourth Int'l Workshop on Testing Database Systems (DBTest'11)*, June 2011.
[64] G. M. Kapfhammer, "Towards a Method for Reducing the Test Suites of Database Applications". Proc. *IEEE 5th Int'l Conf. on Software Testing, Verification and Validation (ICST'12)*, pp. 964-965, Apr. 2013.
[65] F. Haftmann, D. Kossmann and A. Kreutz. "Efficient regression tests for database applications", *Proc. 2nd Conf. on Innovative Data Systems Research*, pp. 95-106, 2005.
[66] F. Haftmann, D. Kossmann and E. Lo, "A framework for efficient regression tests on database applications", *The VLDB Journal*, vol. 16, no. 1, pp. 145-164, Jan. 2007.
[67] V. Sharma and A. P. Agrawal, "Regression Test Case Selection for Testing Database Applications. *International Journal of Innovative Technology and Exploring Engineering*, vol 3, no. 1, pp. 212-216, Jun. 2013.
[68] R. A. Haraty, N. Mansour, B. Daou, "Regression test selection for database applications", *Advanced Topics in Database Research, vol. 3*, K. Siau ed, pp. 141-165, Idea Group Publishing, 2004.
[69] D. Willmor and S. M. Embury, "A safe regression test selection technique for database-driven applications", *Proc. 21st IEEE Int'l Conf. on Software Maintenance (ICSM'05)*, pp. 421-430, 2005.
[70] B. Daou, R. A. Haraty, N. Mansour. "Regression Testing of Database Applications", *Proc. ACM symposium on Applied computing (SAC'01)*, pp. 285-289, 2011.
[71] E. Rogstad, L. Briand, R. Dalberg, M. Rynning and E. Arisholm, "Industrial Experiences with Automated Regression Testing of a Legacy Database Application", *Proc. 27th IEEE International Conf. on Software Maintenance (ICSM'11)*, pp. 362-371, 2011.
[72] E. Rogstad, L. Briand and R. Torkar, "Test case selection for black-box regression testing of database applications", *Information and Software Technology*, Vol. 55, no. 10, pp. 1781-1795, Oct. 2013.
[73] A. C. B. Loureiro, C. G. Camilo-Junio, L. T. Queirozz, C. L. Rodrigues, P. Leitao-Junior and A. M. R. Vincenzik", "Shrinking a Database to Perform SQL Mutation Tests Using an Evolutionary Algorithm", *Proc. IEEE Congress on Evolutionary Computation (CEC'13)*, pp. 20-23, 2013.
[74] B. Glavic and K. Dittrich, "Data Provenance: A Categorization of Existing Approaches", *Proc. Datenbanksysteme in Business, Technologie und Web (BTW'07)*, pp. 227-241, 2007.
[75] Y.L. Simmhan, B. Plale and D. Gannon, "A Survey of Data Provenance in e-Science". *ACM SIGMOD Record*, vol. 34, no. 3, pp. 31-36, Sept. 2005.
[76] R. Ikeda and J. Widom, "Data Lineage: A Survey", Technical report, Stanford University, 2009.
[77] T. Heinis and G. Alonso, "Efficient Lineage Tracking for Scientific Workflows". *Proc. 2008 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'08)*, pp. 1007-1018, 2008.
[78] Y. Cui and J. Widom, "Lineage tracing for general data warehouse transformations", *The VLDB Journal*, vol. 12 no. 1, pp. 41-58, May 2003.
[79] B. Glavic, "Big Data Provenance: Challenges and Implications for Benchmarking", *Proc. 2nd Intl. Workshop on Big Data Benchmarking (WBDB'12)*, 2012.
[80] P. Buneman, S. Khanna, and W.C. Tan, "Why and Where: A Characterization of Data provenance", *Proc. 8th Intl. Conf. on Database Theory (ICDT'01)*, pp. 316-330, 2001.
[81] B. Glavic, "Perm: Efficient Provenance Support for Relational Databases", PhD dissertation, University of Zurich, 2010.
[82] Y. Cui and J. Widom, "Practical Lineage Tracing in Data Warehouses", *Proc. 16th Intl. Conf. on Data Engineering (ICDE'00)*, pp. 367-378, 2000.
[83] Y. Cui, "Lineage Tracing in Data Warehouses". PhD dissertation, Stanford University, 2002.
[84] B. Glavic and G. Alonso, "Perm: Processing Provenance and Data on the same Data Model through Query Rewriting", *Proc. 25th Intl. Conf. on Data Engineering (ICDE'09)*, pp. 174-185, 2009.

[85] B. Glavic and G. Alonso, "Provenance for Nested Subqueries", *Proc. 12th Intl. Conf. on Extending Database Technology (EDBT '09)*, pp. 982-993, 2009.
[86] B. Glavic, R.J. Miller and G. Alonso, "Using SQL for Efficient Generation and Querying of Provenance Information", *In Search of Elegance in the Theory and Practice of Computation*, V. Tannen, L. Wong, L. Libkin, W. Fan, W.C. Tan and M. Fourman, eds. pp 291-320, Lecture Notes in Computer Science, vol. 8000, Springer Verlag, 2013.
[87] M. Harman and B. F. Jones, "Search-based software engineering", *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, Dec. 2001.
[88] M. Harman, S. A. Mansouri and Y. Zhang, "Search-based software engineering: Trends, techniques and applications". *ACM Computing Surveys*, vol. 45, no. 1, Nov. 2012.
[89] P. McMinn, "Search-based software test data generation: a survey", *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, Jun. 2004.
[90] P. McMinn, "Search-Based Software Testing: Past, Present and Future", *Proc. IEEE Fourth Int'l Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 153-163, 2011.
[91] N. Tracey, J. Clark, K. Mander and J. McDermid, "An automated framework for structural test-data generation", *Proc. Int'l Conf. on Automated Software Engineering (ASE'98)*, pp. 285-288, 1998.
[92] P. McMinn and M. Holcombe, "Evolutionary testing using an extended chaining approach", *Evolutionary Computation*, vol. 14, no. 1, pp. 41-64, 2006.
[93] A. Baresel, "Automating structural tests using evolutionary algorithms", Master's Thesis, Humboldt University of Berlin, Germany, 2000.
[94] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing". *Software Testing, Verification and Reliability*, vol. 23, no. 2, pp. 119-147, Mar. 2013.
[95] W. Kim, "On Optimizing an SQL-Like Nested Query". *ACM Transactions on Database Systems*, vol. 7, no. 3, pp. 443-469, Sep. 1982.
[96] R. A Ganski and H. K. T. Wong, "Optimization of Nested SQL-Queries Revisted". *Proc. ACM SIGMOD Int'l Conf. on Management of data*, pp. 23-33, 1987.
[97] Transaction Performance Council, *The TPC Benchmark™H (TPC-H)*, http://www.tpc.org/tpch/ [30 Jan. 2015].
[98] J.H. Andrews, L.C. Briand and Y. Labiche, "Is mutation an appropriate tool for test-ing experiments?", *Proc. 27th Intl Conf. on Software Engineering (ICSE'05)*, pp. 402-411, May 2005.
[99] R. Just, D. Jalali, L. Inozemtseva, M.D. Ernst, R. Holmes and Gordon Fraser, "Are Mutants a Valid Substitute for Real Faults in Software Testing?", *Proc. 22nd ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering (FSE'14)*, pp. 654-665, Nov. 2014.
[100] R. Just, F. Schweiggert and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler", *Proc. Intl. Conf. on Automated Software Engineering (ASE'11)*, pp. 612-615, Nov. 2011.
[101] R. Just, "The Major mutation framework: Efficient and Scalable Mutation Analysis for Java", *Proc. Intl. Symp. on Software Testing and Analysis (ISSTA'14)*, pp. 433-436, Jul. 2014.
[102] Y.S. Ma, J. Offutt and Y.R. Kwon, "MuJava: An Automated Class Mutation System", *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97-133, Jun. 2005.
[103] R. Just, G. M. Kapfhammer and F. Schweiggert, "Using conditional mutation to increase the efficiency of mutation analysis", *Proc. Intl. Workshop on Automation of Software Test (AST'11)*, pp. 50-56, May 2011.
[104] R. Just, G. M. Kapfhammer and F. Schweiggert, "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis", *Proc. Intl. Symp. on Software Reliability Engineering (ISSRE'12)*, pp. 11-20, Nov. 2012.
[105] R. Just, M. D. Ernst and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states", *Proc. Intl. Symp. on Software Testing and Analysis (ISSTA)*, pp. 315-326, Jul. 2014.

Javier Tuya is a Professor at the University of Oviedo, Spain, where he is the research leader of the Software Engineering Research Group. He received his PhD in Engineering from the University of Oviedo in 1995. He is Director of the Indra-Uniovi Chair, member of the ISO/IEC JTC1/SC7/WG26 working group for the recent ISO/IEC/IEEE 29119 Software Testing standard and convener of the corresponding AENOR National Body working group. His research interests in software engineering include verification & validation and software testing for database applications and services. He is a member of the IEEE, IEEE Computer Society, ACM and the Association for Software Testing (AST).

Claudio de la Riva is an assistant professor at the University of Oviedo, He is a member of the Software Engineering Research Group (GIIS, giis.uniovi.es). He obtained his PhD in Computing from the University of Oviedo. His research interests include software verification and validation and software testing mainly focused on testing database applications and services. He is a member of ACM

María José Suárez-Cabal is an assistant professor at the University of Oviedo, Spain, and is a member of the Software Engineering Research Group (GIIS, giis.uniovi.es). She obtained her PhD in Computing from the University of Oviedo in 2006. Her research focusses on software testing, and more specifically on testing database applications.

Raquel Blanco is an assistant professor at the University of Oviedo, Spain, and is a member of the Software Engineering Research Group (GIIS, giis.uniovi.es). She obtained her PhD in Computing from the University of Oviedo in 2008. Her research focusses on software testing, mainly on testing database applications and the user-database interaction.