This is the submitted version of the following article: Full predicate coverage for testing SQL database queries, which published in final form at Software Testing, Verification and Reliability, 20(3) 237-288, September 2010. DOI: 10.1002/stvr.424. Submitted: Feb, 27th, 2009, Accepted: Nov, 2th, 2009.

Full predicate coverage for testing SQL database queries

Javier Tuya *, Mª José Suárez-Cabal and Claudio de la Riva

Department of Computer Science, University of Oviedo,

Campus of Viesques, s/n, 33204 Gijón (SPAIN)

SUMMARY

In the field of database applications a considerable part of the business logic is implemented using a semi-declarative language: the Structured Query Language (SQL). Because of the different semantics of SQL compared to other procedural languages, the conventional coverage criteria for testing are not directly applicable. This paper presents a criterion specifically tailored for SQL queries (SQLFpc). It is based on Masking Modified Condition Decision Coverage (MCDC) or Full Predicate Coverage and takes into account a wide range of the syntax and semantics of SQL, including selection, joining, grouping, aggregations, subqueries, case expressions and null values. The criterion assesses the coverage of the test data in relation to the query that is executed and it is expressed as a set of rules that are automatically generated and efficiently evaluated against a test database. The use of the criterion is illustrated in a case study which includes complex queries.

KEY WORDS: software testing; database testing, MCDC, Full Predicate Coverage, SQL

* Correspondence to: Javier Tuya, Departamento de Informática, Universidad de Oviedo, Campus de Viesques s/n, E-33207 Gijón (SPAIN)
Tel.: (34) 985 182 049, FAX: (34) 985 181 986
E-mail: tuya@uniovi.es

Contract/grant sponsor: Department of Science and Innovation (Spain) and ERDF Funds; contract/grant number: TIN2007-67843-C06-01
Contract/grant sponsor: Government of the Principality of Asturias; contract/grant number: CN-07-168
Contract/grant sponsor: Government of Castilla-La Mancha; contract/grant number: PAC08-121-1374

1. INTRODUCTION

Database applications involve the management of large amounts of data stored and organized in many tables. Although there have been developments in object oriented databases and more recently in eXtensible Markup Language (XML) databases, most applications still maintain the data using Relational Database Management Systems (DBMS) that provide a high performance and a high degree of scalability and dependability. Different solutions to manage the data have been developed (such as persistence systems or object/relational mappings). However, the Structured Query language (SQL) [1] is still widely used, especially when its full expressive power is needed [2]. On the other hand, SQL is a semi-declarative language which embodies a complex processing in each query: A single SELECT query may combine data from several tables, select data based on a logical expression, group the selected data according to some criteria and perform a further selection and ordering. Moreover, the three-valued logic [3] of the logical expressions adds additional semantic complications. The testing of such query requires preparing the input (which is the database itself and therefore may involve many tables) and checking the output, which is another table-like structure.

One of the major fields of study in coverage criteria for testing is related to coverage of the source code using different approaches (e.g. data-flow or control-flow). Control-flow criteria range from path or branch coverage to more sophisticated criteria to thoroughly assess the adequacy of tests according to their logical decisions (denominated as ‘logic testing’ by Kaminski et al. [4]). These criteria are a powerful tool to evaluate the adequacy of test suites and to assist in the development or completion of test cases.

In database applications the software under test interacts with the database in single interaction points where the program passes control to the DBMS that executes an SQL command and updates the database or retrieves data from it. In this case, the usual control-flow based criteria may be valid to assess the adequacy of the code that creates the SQL query and process the results. However, as a very important part of the business logic is implemented in the SQL query, if the input data and the query are not taken into account in the test design, much of the complexity of the logic decisions taken by the application may be concealed from the tests. One of the most difficult, time intensive and error prone activities when testing applications with databases is the preparation of a suitable set of data for covering, as much as possible, the different situations that may occur and have to be considered by the queries. As when testing an imperative program, a coverage criterion that allows assessing the adequacy of the test data in relation to the query that processes such data may be a valuable help to the tester in developing higher quality test databases. The main difference is that in this case the test input is a database with a complex structure and many rows and the program under test is procedural. This is the issue that is addressed in this paper.

The evaluation of the coverage of SQL queries according to a given criterion can be potentially used in a number of different scenarios. During the development, a small test database may be created from scratch in order to fulfill the coverage of test data against the query. A controlled experiment previously conducted by the authors [5] has shown that when the user develops a test database assisted by a coverage criterion, it leads to tests that are able to detect more faults in the query than if he/she is not guided by the criterion. However, creating new test databases for each query is very time consuming. Therefore, a commonly used approach is to begin with a previously populated database and then complete it with meaningful data to test the query. In this case the evaluation of the coverage will assist the tester in completing the test inputs. Another scenario is that of evaluating the extent to which a given test suite (which usually contains many test cases for each database load) exercises several queries of the application. In this case the queries that are executed can be collected using vendor specific DBMS tracing tools or using external software like p6spy[1] and then evaluating their coverage. This information is valuable for analysis and as feedback in order to complete the test cases.

In a previous work the use of a criterion like Modified Condition Decision Coverage (MCDC) for testing SQL queries was suggested and informally presented [6]. This article focuses on the complete definition and automation of a criterion named SQLFpc that measures the coverage of the test data in relation to the query that is executed. The primary contributions of this paper are:

- The development of a coverage criterion for SQL based on Full Predicate Coverage, defined by Offutt et al. [7] or masking MCDC, defined by Chilenski [8], considering the specific semantics of a wide range of the SQL syntax and database schema constraints. The criterion identifies the requirements that have to be satisfied by the test data used by SELECT queries, including JOIN, WHERE, HAVING and GROUP BY clauses, aggregate functions, subqueries and case expressions as well as the handling of missing information (null values).
- A complete description of how the test requirements are expressed as a set of coverage rules that is obtained by applying successive transformations on the query under test. The coverage rules are executable and able to determine whether the different situations expressed by the test requirements are covered by the test data.
- The completely automated generation and evaluation of the coverage rules, which are implemented in a set of tools, and the efficient evaluation of the coverage, even for large databases. As the rules are also SQL queries they take advantage of all performance improvements implemented in commercial DBMS.

[1] p6spy is an open source Java tool that intercepts and logs all database statements that use JDBC, available at http://www.p6spy.com/.

- The application of the coverage criterion to a case study including complex queries taken from *Compiere*, which is an open source Enterprise Resource Planning (ERP) application. The case study shows that as the test database contains more rows that increase the SQLFpc coverage, they are able to reveal more faults in the query (measured in terms of mutation score).

The paper is organised as follows: Section 2 presents an overview of basic notation, the relational model and the MCDC and Full Predicate Coverage criteria. The core of the article begins in Section 3 that describes the coverage rules for WHERE and JOIN clauses when considered in isolation. Section 4 describes the rules for their combinations and Section 5 addresses the rest of SQL clauses. Section 6 summarizes all rules presented so far and outlines the tool support available. Finally, Section 7 uses the coverage rules in a case study, Section 8 presents the related work and Section 9 concludes.

2. BACKGROUND AND NOTATION

This section introduces the notation that will be used in subsequent sections and provides some background about the relational model (Section 2.1), its basic operators (Section 2.2) and the MCDC coverage criterion (Section 2.3).

2.1. The relational model ま Basic components

The relational model was first developed by Codd [9] and defines the foundations of data storage and querying that is implemented in today's commercial database management systems. The notation used in this paper is that presented by the author in the second version of the relational model [10], from now on referred to as RM/V2, with some adaptations that are needed for subsequent sections.

Relations and attributes: Given a set A of attributes A_1, A_2, ⋯, A_n, with domains D_1, D_2, ⋯, D_n respectively (attributes are denoted using uppercase letters), a relation R (also named R-table) is a subset of the Cartesian product $D_1 \times D_2 \times \cdots \times D_n$. The relation R is represented as $R(A_1, A_2, \ldots A_n)$ 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.

Base and derived attributes: In the RM/V2 each relation is defined in terms of attributes. In SQL an attribute may be derived from a function or expression. An attribute in a relation is said to be a *derived attribute* if it is calculated after applying a function or expression over other attributes in the relation. The attribute is said to be a *base attribute* if its value does not depend on any other attributes. The set of base attributes from which a given derived attribute F is obtained is denoted as battrs(F).

Missing information: The RM/V2 extends the previous version by defining a four-valued logic. There is a semantic distinction between missing but applicable (A-mark) and inapplicable (I-mark). In commercial DBMS, however, this semantic distinction is not implemented, and in all cases A-marks and I-marks are indicated as NULL. In order to make the handling of missing information compatible with SQL the following definitions are stated:

- An attribute A_i is *nullable* iff it has been declared in the database schema without the IS NULL constraint. The Boolean predicate nullable(A_i) is true iff the attribute A_i is nullable.
- The value of an attribute A_i at a given tuple is *null* iff it is not known (missing) at the moment in which A_i is evaluated. The *null-check predicate* is a boolean predicate nl(A_i) that is true at the tuples in which A_i is null.

Predicates: Predicates are three-valued, so as their result may be true, false or unknown. Each predicate may be composed of an arbitrary logical expression on attributes. The following definitions are applied taking into account the existence of the logical operators ($\wedge$, $\vee$, $\neg$):

- A *base predicate* is a three-valued logical expression that contains no logical operators[2].
- A *derived predicate* is a three-valued logical expression that is composed of predicates (either base or derived) and one or more logical operators.

The set of all base predicates included in a given predicate p is denoted as bpreds(p). The set of all base attributes in a given predicate p is denoted as battrs(p).

For example, the derived predicate p := $A_1+A_2=0 \wedge nl(A_3)$ is composed of two base predicates: bpreds(p)={$A_1+A_2=0$, $nl(A_3)$}. The first one contains two base attributes: battrs($A_1+A_2=0$)={A_1,A_2}. The second base predicate includes a null-check predicate over a single base attribute: battrs($nl(A_3)$)={A_3}.

2.2. The relational model まBasic operations

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←rve where rve denotes a *relation-valued expression* (RVE) and Z is the name for the relation obtained when applying the relation-valued expression. The basic relational operators and the corresponding relation-valued expressions are now described:

Select Operator: The Select operator employs a single relation R(A) as operand and generates as a result a relation (Z) that contains some of the complete rows that the operator contains according to a criterion specified using a logical predicate p on the attributes of A. The select operator returns only those tuples that make true the predicate p(A).

$Z \leftarrow R[p(A)]$, in SQL: SELECT * FROM R WHERE p(A)

Join Operator: The join operator employs two relations R(A) and S(B) as its operands and generates as a result a relation (Z) that contains rows of R(A) concatenated with rows of S(B), but only where the condition specified by a predicate p(A,B) is found to hold true. The predicate p is named *join predicate*:

$Z \leftarrow R[p(A,B)]S$, in SQL: SELECT * FROM R INNER JOIN S ON p(A,B)

Note that joins are defined in terms of relations. Each of the constituent relations can be either a derived relation (obtained from a join operator) or a base relation (relation that is internally represented by stored data, i.e. a table in a DBMS).

Inner and Outer join operators: The previous operator is also named *inner join*. Three other different types of join are defined (namely *left*, *right* and *full outer join*). The left outer join is the union of the tuples obtained by applying the inner join operator and the *left outer increment* (LOI). The RM/V2 defines LOI as follows: ゆick out those tuples from R whose comparand values in the comparand column do not participate in the inner join, and append to each such tuple a tuple of nothing but missing values and of size compatible with Sめ The right outer join is symmetrically defined in terms of the *right outer increment* (ROI). The full outer join (also named symmetric outer join) is defined as the union of the inner join, the LOI and the ROI. The notation used in this article for representing the relation-valued expressions for joins is:

$$R[p(A,B)]^{JT}S$$

Where JT is a label {I, L, R, F} that denotes the join-type (inner, left outer, right outer and full outer join, respectively). When a join is referred, regardless of its join-type, the label JT may be omitted.

[2] Base predicates are denoted as clauses by Offut et al [7]. In this paper the term base predicate is used instead for consistence with other terms such as base tables and derived tables used in the RM/V2. The term clause will be reserved to refer to clauses as specified in the SQL Standard.

Framed Relations: A frame 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 framing is performed in SQL by the GROUP BY clause and is denoted as:

$$Z \leftarrow R /// G, \text{ in SQL: SELECT G,F FROM R GROUP BY G}$$

Where R is a relation of attributes A, G is the set of grouping attributes, $G \subseteq A$, and F is a set of derived attributes in the form f(A). Each of the functions f (aggregate functions) computes a statistical value on all tuples that belong to each subrelation. The result of the framing is a relation containing a single tuple for each subrelation.

An additional select operator may be applied to the framed relation in the form (HAVING clause in SQL):

$$Z \leftarrow R /// G\ [q(G,F)], \text{ in SQL: SELECT G,F FROM R GROUP BY G HAVING q(G,F)}$$

2.3. MCDC and Full Predicate Coverage

Modified Condition Decision Coverage (MCDC), defined in the RTCA/DO-178B standard [11], is a coverage criterion which has been demonstrated as representing a good balance of test-set size and fault detecting ability [12,13], studied in depth in the literature [14-17] and used for test suite reduction and prioritization [18]. MCDC requires 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. Some of the advantages of MCDC over other criteria for logic testing are that it needs a small test size and that it does not require a predicate to be in any particular syntactic format, such as Disjunctive Normal Form (DNF). However, the sensitivity of MCDC to specific logic faults is lower than other criteria and it may still miss some faults that are always detected by other criteria as shown by Kaminski et al. [4] and Yu and Lau [13].

Consider the decision $p=(a \wedge b) \vee (c \wedge d)$ composed of four boolean conditions (a,b,c,d). To satisfy MCDC, for each condition a pair of test cases (also called *test points*) is generated. For instance, a pair that satisfies the criterion with respect to condition a is (0,1,1,0) and (1,1,1,0), because when a flips while all the other conditions do not change, the result of the decision changes. These pairs of test points are called *independence pairs*.

Chilenski [8] has elaborated three different forms of MCDC: Unique-Cause, Unique-Cause Masking and Masking. Ammann et al. [19] and Ammann and Offutt [20] have presented a complete comprehensive set of criteria: Active Clause Coverage (ACC) and Inactive Clause Coverage (ICC). Three forms of ACC are presented: General, Correlated and Restricted. Masking MCDC, which is equivalent to Correlated ACC, is the most flexible form which allows the other conditions to change while showing the independence effect of each condition. Consider the previous example: A pair (0,1,1,0) and (1,1,0,1) shows the independence effect of a, although c and d have been changed, because the subexpression $c \wedge d$ has not changed.

Full Predicate Coverage, defined by Offutt et al. [7] is a form of masking MCDC developed for testing the conditions that trigger transitions in state-based specifications. It requires that for each predicate p on each transition and for each test clause (test condition) in each predicate, the set of test cases must include tests that cause each clause c_i in p to *determine* the value of p. A test clause c_i is said to determine p if the remaining minor clauses have values such that changing the truth value of c_i changes the truth value of p.

Consider the predicate previously used $p=(a \wedge b) \vee (c \wedge d)$. The procedure described by Offutt et al. [7] to find values that satisfy full predicate coverage is based on walking the parse tree of the predicate from the bottom to the root. The parse tree for predicate p is depicted in Figure 1. For instance, in order to let clause a determine the output, the subpredicate $(a \wedge b)$ requires the sibling b to be true (because the expression contains an AND). Then walking up the tree, the sibling

subpredicate (c∧d) must be false (because the expression contains an OR). Note that any combination for the values of clauses c and d which makes the subpredicate (c∧d) false is allowed because the criterion used is masking MCDC.

These definitions are general enough that they can be adapted to the particularities of SQL. Then, they will constitute the basic definition of the coverage criterion to be used in the rest of the paper.

3. BASIC COVERAGE RULES

This section presents the procedure for evaluating the coverage of SQL queries (represented as relation-valued expressions), which consists in obtaining a set of coverage rules (also represented as relation-valued expressions) that can be applied against the test input (relations in the database). Firstly the concept of coverage rules is presented (Section 3.1). Next the procedure for creating the coverage rules for most common operators is detailed, starting with select operators (Section 3.2). Due to the complexity of join operators, their corresponding section is split into Section 3.3 which deals with individual joins and Section 3.4 which deals with multiple joins (nested joins).

3.1. Coverage rules

In the context of database testing, and using the notation presented in the previous section, the smallest unit to be tested is an RVE (in SQL, a query). A *test input* is a set of base relations (tables populated with data) and the *test output* is another relation (the output of the query) resulting after applying the relational assignment (executing the query).

The process of designing a given test input for a given query using a given coverage criterion (such as full predicate) begins with the selection of the *test conditions* to be tested. For each one, a set of *test points*[3] is selected on the basis of the given *test point requirements* imposed by the coverage criterion. Then the test points are combined into a *test input* which along with the desired output constitutes the *test case*. Consider a simple query Q represented by the following RVE:

$$Z \leftarrow T[A_1=1 \wedge A_2=0], \qquad \text{in SQL: SELECT * FROM T WHERE A1=1 AND A2=0}$$

Consider the first test condition (A_1=1). Assuming that the attribute A_1 is not nullable, the criterion may state two test point requirements: (1) select a test point such that the test condition is true: A_1=1 and (2) another one such that the test condition is false: ¬(A_1=1), with A_2=0 in both test points. Each test point is a tuple which satisfies each of the requirements, for instance tuple {(1,0)} and tuple {(4,0)}, respectively. These test points are combined in a relation composed of the above tuples which constitutes the test input: {(1,0), (4,0)}.

If the goal is to evaluate the coverage of a given test input with respect to the query a simple approach is to formalize the test point requirements into a predicate and then check whether the predicate holds. In the example, consider a given test input {(1,0), (4,0)}. The two test point requirements are stated in the form of predicates (1) A_1=1∧A_2=0 and (2) ¬(A_1=1)∧A_2=0 that would be evaluated against the test input {(1,0), (4,0)}. It is clear that this test input satisfies each of the predicates, and then both test point requirements are covered.

Since the goal of this paper is to describe how to evaluate the coverage in SQL queries some important issues must be resolved:

[3] The term test point is commonly used in the MCDC literature as synonymous of test case. In the context of database testing there is a significant difference, because a single test case is obtained by filling a set of relations that usually include many test points. Therefore each single test case may exercise many different situations represented by the test points.

1. How to interpret the full predicate criterion in order to tackle the particularities of SQL. That will be accomplished by determining what the test conditions are and taking into account the three-valued logic, the semantics of the different predicates and operators (e.g. select and joins) and the combinations of different operators that constitute a query.
2. How to elaborate the test point requirements, taking as the only sources of information the query and the database schema. This will be accomplished by transforming the original query into queries that express each test point requirement. This kind of queries is called coverage rules. Each coverage rule is associated with a single test point requirement.
3. How to evaluate each coverage rule in order to determine whether its test point requirement holds (the rule is covered) by a given test input. That is immediate because each coverage rule is a query that can be executed against the test input.

The interpretation and elaboration of the test point requirements will be detailed in the following sections. Now, some basic definitions are presented:

Definition 1. A *coverage rule* (Δ) is an RVE constructed in order to evaluate the fulfillment of a given test point requirement. A coverage rule is covered (its test point requirement holds) if the relational assignment of the rule ($Z \leftarrow \Delta$) results in a non empty relation ($Z \neq \varnothing$).

In the above example, the two test point requirements were expressed as two predicates (1) and (2), respectively, which are embedded in two select operators to constitute the coverage rules Δ_T and Δ_F, respectively.

$$\Delta_T\,(T[p]) := T[A_1=1 \wedge A_2=0] \quad \text{and} \quad \Delta_F\,(T[p]) := T[\neg(A_1=1) \wedge A_2=0]$$

It is easy to check that each of the coverage rules will result in a non empty relation when evaluated against the input {(1,0), (4,0)}.

Definition 2. A *coverage rule transformation* (Φ) is a transformation from an RVE (or part) into another RVE (or part). Given a query expressed as an RVE, a coverage rule is obtained by successively applying one or more coverage rule transformations to the query or part of it.

In the following subsections the set of transformations needed to generate rules that fulfill the full predicate coverage criterion for select and join operators is detailed.

3.2. Coverage rules for select operators

Introductory example 1. Let p be a predicate in an RM/V2 select operator in the form $p=p_1 \wedge c_3 \wedge c_4$ where $p_1=c_1 \vee c_2$ (such that p_1 is a derived predicate and c_1 to c_4 are base predicates). In order to get test cases to fulfill the full predicate coverage for the test condition c_1 a pair of test points is needed in order to let c_1 determine the output of p. By following the procedure described in Section 2.3, it requires that c_1 takes true and false, its sibling predicate c_2 be false (as the logical operator is or) and walking up the parse tree the siblings of p_1 (parent of c_1) be true (as the logical operator is and). Then the constraints that must be satisfied for each test point (test point requirements) are respectively:

$$c_1 \wedge \neg(c_2) \wedge (c_3 \wedge c_4)$$

$$\neg c_1 \wedge \neg(c_2) \wedge (c_3 \wedge c_4)$$

Moreover, because of the use of a three-valued logic, if c_1 is nullable, there is another important value to test. An additional test point requirement is needed (note that c_1 determines the output if its value flips over true and null):

$$nl(c_1) \wedge \neg(c_2) \wedge (c_3 \wedge c_4)$$

The predicates $\neg(c_2)$ and $(c_3 \wedge c_4)$, respectively are to be named the *Sibling Independence Predicates* (SIP) with respect to c_1 and p_1 respectively, and $\neg(c_2) \wedge (c_3 \wedge c_4)$ the *Independence*

Predicate (IP) with respect to the test condition c_1. In general, the procedure to obtain the test point requirements for a given test condition c_i consists in obtaining the SIP with respect to c_i and then obtaining the IP with respect to its parent and so on, until the root is reached.

Definition 3. (Evaluation of select predicates) Let R[p(A)] be a select operator. The test conditions for the select operator are each of the base predicates $c_i \in bpreds(p)$. At a given tuple in $Z \leftarrow R[p(A)]$, each test condition may evaluate to one of the following values: True (T), False (F) or Null (N).

Definition 4. (Coverage transformations for select predicates) Let R[p(A)] be a select operator. Let p be the select predicate in the form (p_1 lop p_2 lop⋀lop p_n), where lop is a logical operator $\{\wedge,\vee\}$ and p_i are either base or derived predicates. The Sibling Independence Predicate of each of the sub-predicates p_i is defined as:

$$SIP(p_i) := \begin{cases} \bigwedge_{j \neq i} p_j & \text{if lop is } \wedge \\ \bigwedge_{j \neq i} \neg p_j & \text{if lop is } \vee \end{cases}$$

The Independence predicate (IP) of p with respect to a test condition $c_i \in bpreds(p)$ is the predicate that must be satisfied for all test points that state that c_i determines p. It is determined recursively as a conjunction of the SIP of c_i and the IP of its parent:

$$IP(c_i) := SIP(c_i) \wedge IP(parent(c_i))$$

The coverage rule transformations to attain full predicate of p with respect to each test condition c_i are defined for each of the possible evaluations of the select predicate as:

$$\Phi_T(p,c_i) := c_i \wedge IP(c_i)$$

$$\Phi_F(p,c_i) := \neg c_i \wedge IP(c_i)$$

$$\Phi_N(p,c_i) := nl(c_i) \wedge IP(c_i)$$

However in Φ_N the test condition c_i may reference more than one nullable attribute. In order to maintain the independence principle for each one, the Φ_N transformation is redefined to take into account each $A_k \in battrs(c_i)$ such that nullable(A_k) holds:

$$\Phi_N(p,c_i,A_k) := nl(A_k) \wedge IP(c_i)$$

Non independent conditions. One of the advantages of Full Predicate Coverage or Masking MCDC over other MCDC versions is that it is less sensitive to coupled conditions (non independent conditions). However, the introduction of the Φ_N transformations may produce more situations in which non independent conditions cause impossible test points. Consider the predicate p = a>1 ∧ a<5. The transformation Φ_N(p, a>1, a) produces nl(a) ∧ a<5, which generates a coverage rule impossible to be fulfilled because if nl(a) is true, then a<5 is undefined and then the result is undefined. This is similar to the case of unreachable paths in an structured program, which prevents having a test set for achieving 100% coverage. To be able to exercise the situation in which nl(a) holds, the predicate a<5 should be removed from the transformed RVE. This suggests the redefinition of Φ_N as explained below:

Definition 5. (Null Reduction Transformation) Let R[p(A)] be a select predicate and A_k a base attribute in p. The Null Reduction transformation of p with respect to A_k denoted as NR(p,A_k) transforms p in such a way that every base predicate c_i in p such that $A_k \in battrs(c_i)$ is removed from p.

Then the Φ_N is redefined again for each $A_k \in battrs(c_i)$ such that nullable(A_k) holds by applying the null reduction transformation to the Independence Predicates:

$$\Phi_N(p,c_i,A_k) := nl(A_k) \wedge NR(IP(c_i),A_k)$$

Definition 6. (Coverage rules for select predicates) Let R[p(A)] be a select operator, and c_i each of the base predicates of p. The set of coverage rules Δ_T, Δ_F and Δ_N are defined as:

$$\forall c_i \in bpreds(p): \Delta_T(p,c_i) := R[\Phi_T(p,c_i)] \quad (1)$$

$$\forall c_i \in bpreds(p): \Delta_F(p,c_i) := R[\Phi_F(p,c_i)] \quad (2)$$

$$\forall c_i \in bpreds(p), \forall A_k \in battrs(c_i) \mid nullable(A_k): \Delta_N(p,c_i,A_k) := R[\Phi_N(p,c_i,A_k)] \quad (3)$$

It must be noted that if c_i is a null-check predicate in the form c_i=nl(A) or c_i=¬nl(A), respectively the transformation Φ_N is applied instead of Φ_T or Φ_F, respectively. Also, when a predicate is in the form $c_1 \wedge \ldots \wedge c_n$ or $c_1 \vee \ldots \vee c_n$, where each c_i is a base predicate, the rules Δ_T or Δ_F, respectively are generated only for the first base predicate in order to avoid duplicate rules.

Boundaries. Conditions that involve a relational operator (e.g. a>b) are a well known source of potential faults when the relational operator is wrong. In order to detect this kind of faults, instead of placing test cases in the equivalence classes a>b and ¬(a>b), the test cases are placed in the boundary values. Then instead of having two test points there will be three a=b+1, a=b and a=b-1. These are the *operator assurance extensions* defined by Chilenski [8]. In the context of the FPC criterion, it will be defined as follows:

Definition 7. (Coverage rules and transformations for checking boundary values) Let c_i be a base predicate in the form a rop b where rop is a relational operator in {=, ≠, >, ≥,<, ≤}. If the domain of both attributes is numeric, then the coverage transformations Φ_T and Φ_F in Definition 4 are replaced by:

$$\Phi_{B+}(p,c_i) := (a \text{ rop } b+1) \wedge IP(c_i)$$

$$\Phi_{B=}(p,c_i) := (a \text{ rop } b) \wedge IP(c_i)$$

$$\Phi_{B-}(p,c_i) := (a \text{ rop } b-1) \wedge IP(c_i)$$

And the corresponding rules Δ_T and Δ_F in Definition 6 are redefined into Δ_{B+}, $\Delta_{B=}$, Δ_{B-}, to use the above transformations, respectively.

Select Operators after Framing. A select operator may appear after a framed relation (HAVING clause in SQL) in the form R /// G [q(B,F)]. In this case, the coverage rule transformations for the predicate are obtained in the same way as has been presented above, considering that F consists of derived attributes which include the SQL aggregate functions (count, sum, min, max, avg).

Example 1. Consider the query R[q] defined as: R[(a='x' ∨ nl(a)) ∧ b>c]. The coverage rules obtained after applying the above transformations (definitions 6 and 7) to each test condition are:

$\Delta_T(q, a=$ ▯▯$):= R[a='x' \wedge \neg nl(a) \wedge b>c]$
$\Delta_F(q, a=$ ▯▯$):= R[\neg a='x' \wedge \neg nl(a) \wedge b>c]$
$\Delta_N(q, a=$ ▯▯a$) := R[nl(a) \wedge b>c]$
$\Delta_N(q, nl(a), a) := R[nl(a) \wedge b>c]$
$\Delta_{CB+}(q, b>c) := R[b=c+1 \wedge (a='x' \vee nl(a))]$
$\Delta_{CB=}(q, b>c):= R[b=c \wedge (a='x' \vee nl(a))]$
$\Delta_{CB-}(q, b>c) := R[b=c-1 \wedge (a='x' \vee nl(a))]$
$\Delta_N(q, b>c, b) := R[nl(b) \wedge (a='x' \vee nl(a))]$
$\Delta_N(q, b>c, c):= R[nl(c) \wedge (a='x' \vee nl(a))]$

The test condition a='x' generates three rules. The test condition nl(a) generates Δ_N instead of Δ_T after reducing predicate a='x' and does not generate Δ_F because it has been previously obtained for predicate a='x'. The test condition b>c generates three rules for boundary values and two for nulls (provided that both b and c are nullable). A further processing would remove duplicate rules (because Δ_N is the same for a=▯▯and nl(a)).

3.3. Coverage rules for single join operators

Introductory example 2. Consider a simple full outer join in the form

$Z \leftarrow R[A_1=B_1]^F S$, in SQL: SELECT * FROM R FULL OUTER JOIN S ON A1=B1

In a first approach, the coverage rule transformations presented in Section 3.2 could be applied to the join predicate $A_1=B_1$ in order to generate the coverage rules. However, the semantics of a join operator is completely different than the select operator. Table 1 illustrates the tuples produced as a result of a full outer join on relations R(A)={(11,x), (12,y)} and S(B)={(21,x), (22,z)} The first row is the set of tuples that participate in the inner join (tuples in which $A_1=B_1$), the second one is the left outer increment (tuples in R that do not participate in the inner join, plus missing values in B) and the third one is the right outer increment. But the join predicate evaluates to unknown in the last two rows; however the output is different. This suggests a different way to evaluate join operators in terms of the existence of the inner join, left outer increment and right outer increment instead of the evaluation of the join predicate.

Definition 8. (Evaluation of join operators) Let R[p(A,B)]S be a join operator. The test condition for a single join operator is the operator itself. At a given tuple in $Z \leftarrow R[p(A,B)]S$, each test condition may evaluate to one of the following values: Inner (I) if the tuple participates in the inner join (predicate p is true), Left (L) if the tuple participates in the Left Outer Increment (predicate p is not true) and Right (R) if the tuple participates in the Right Outer Increment (predicate p is not true).

In the previous example, the join is evaluated to each of the possible values, and so it can be said that these tuples cover each of the possible evaluations of the join operator as defined above. Given a join operator the procedure to evaluate the coverage will consist in generating a coverage rule for each of the possible values (I, L or R). The following rule transformations are to be defined:

Definition 9. (Join transformations) Let R[p]S be a join operator denoted as J, and JT in {L,R,I} a label which denotes a join type (note that join type F is not included). The join type transformation $\Phi_J(JT,J)$ transforms the join type of J into the join type specified by the label JT:

$$\Phi_J(JT,R[p]S) := R[p]^{JT}S$$

In the previous example, after applying the join type transformations for L, R and I to the original join the transformations to L and R also return the tuples in inner join, which must be removed. The following additional transformation is needed in order to avoid that:

Let lattrs(p) and rattrs(p) be the attributes of the left and right relations respectively which are referenced by the join predicate p. The outer join select transformations $\Phi_{LOI}(J)$ and $\Phi_{ROI}(J)$ transform the join predicate of the operator into a select predicate which selects only those tuples that compose the LOI and ROI respectively:

$$\Phi_{LOI}(J) := \left(\bigwedge_{A_i \in lattrs(p)} \neg nl(A_i)\right) \wedge \left(\bigwedge_{B_i \in rattrs(p)} nl(B_i)\right)$$

$$\Phi_{ROI}(J) := \left(\bigwedge_{A_i \in lattrs(p)} nl(A_i)\right) \wedge \left(\bigwedge_{B_i \in rattrs(p)} \neg nl(B_i)\right)$$

Definition 10. (Coverage rules for a single join operator) Let R[p]S be a join operator denoted as J. The coverage rules Δ_I, Δ_L and Δ_R select the tuples that belong to the inner join, the left outer increment and the right outer increment, respectively, and are defined as:

$$\Delta_I(J) := \Phi_J(I,J) \quad (4)$$

$$\Delta_L(J) := \Phi_J(L,J)\ [\Phi_{LOI}(J)] \quad (5)$$

$$\Delta_R(J) := \Phi_J(R,J)\ [\Phi_{ROI}(J)] \quad (6)$$

Example 2. Consider the query presented in the introductory example *2*. The three coverage rules for the join operator are:

$\Delta_I(R[A_1=B_1]^F S) := \quad R[A_1=B_1]^I S$
$\Delta_L(R[A_1=B_1]^F S) := \quad (R[A_1=B_1]^L S)[\neg nl(A_1) \wedge nl(B_1)]$
$\Delta_R(R[A_1=B_1]^F S) := \quad (R[A_1=B_1]^R S)[nl(A_1) \wedge \neg nl(B_1)]$

Note that in this case the same rules would be obtained if the join type was different.

Foreign keys. Using the usual terminology in databases, a master-detail relationship relates two tables such that the detail table references the master table. The above coverage rules will be covered if there exist tuples in both master and detail that participate in the join, tuples in the master that do not participate, and tuples in the detail that do not participate. However, some of them may be impossible if there are foreign key constraints. Consider the above example and Table 1; R is the detail and S is the master: if A_1 is a foreign key that references B_1, then the tuple (12,y) in R is not allowed in the database as there is no tuple in B with B_1=y. Therefore, the tuple (2,y,null,null) would never have been obtained in Z. Then the Δ_L is impossible.

Let FK(R,S) be the set of all attributes in R comprising a foreign key referencing some attribute in S. Then the Δ_L rule is impossible if some of the attributes in lattrs(p) are in FK(R,S). The situation is symmetric for Δ_R. Impossible rules are detected and not generated to evaluate the coverage.

Nullable attributes in the join predicate. Consider again the tuples represented in Table 1. No assumptions have been made about the nullability of A_1 and B_1. However, each of them may be nullable even if there is referential integrity. For instance, if A_1 is nullable it is possible to add a tuple like (13,null) in R, which will produce an additional tuple in S: (13,null,null,null). Note that this tuple is also contained in the LOI, but A_1 has a missing value. Two additional rules and transformations must be defined for selecting this kind of tuples.

Definition 11. (Coverage rules and transformations for a single join operator with nullable attributes). Let R[p]S be a join operator denoted as J and p the join predicate with nullable attributes. Let A_k and B_k be each of the the nullable attributes such that $A_k \in$ lattrs(p) and $B_k \in$ rattrs(p) respectively. The *nullable outer join transformations* $\Phi_{NLOI}(J,A_k)$ and $\Phi_{NROI}(J,B_k)$ transform the join predicate of the operator into a select predicate which selects the tuples that compose the LOI or ROI, respectively, and $nl(A_k)$ or $nl(B_k)$ holds respectively:

$$\Phi_{NLOI}(J,A_k) := \left(\bigwedge_{A_i \in lattrs(p)-\{A_k\}} \neg nl(A_i)\right) \wedge nl(A_k) \wedge \left(\bigwedge_{B_i \in rattrs(p)} nl(B_i)\right)$$

$$\Phi_{NROI}(J,B_k) := \left(\bigwedge_{A_i \in lattrs(p)} nl(A_i)\right) \wedge nl(B_k) \wedge \left(\bigwedge_{B_i \in rattrs(p)-\{B_k\}} \neg nl(B_i)\right)$$

In addition to the coverage rules in Definition 10, the following coverage rules Δ_{NL} and Δ_{NR} are also defined:

$$\forall A_i \in lattrs(p) \mid nullable(A_i) : \Delta_{NL}(J,A_i) := \Phi_J(L,J)\ [\Phi_{NLOI}(J,A_i)] \quad (7)$$

$$\forall B_i \in rattrs(p) \mid nullable(B_i) : \Delta_{NR}(J,B_i) := \Phi_J(R,J)\ [\Phi_{NROI}(J,B_i)] \quad (8)$$

Example 3. Consider the query of the introductory example 2, but having referential integrity from A1 to B1 and A1 is nullable. The resulting coverage rules are:

$\Delta_I(R[A_1=B_1]^F S) := \quad R[A_1=B_1]^I S$
$\Delta_{NL}(R[A_1=B_1]^F S, A_1) := \quad (R[A_1=B_1]^L S)\ [nl(A_1) \wedge nl(B_1)]$
$\Delta_R(R[A_1=B_1]^F S) := \quad (R[A_1=B_1]^R S)\ [nl(A_1) \wedge \neg nl(B_1)]$

Note that Δ_L is not generated because of the referential integrity, and a new Δ_{NL} rule is generated for A1 as a result of the Φ_{NLOI} transformation.

3.4. Coverage rules for nested joins

Each participant relation in a join may be either a base relation or a derived relation obtained from a join. Therefore, a set of coverage rules as described in the previous section must be generated for each of the joins. However, the process is not straightforward as illustrated in the following example.

Introductory example 3. Consider the following nested join composed of two joins (1) J_1 on R and S and (2) J_0 on J_1 and T:

$$Y \leftarrow (R[A_1=B_1)]\ ^{L}S)\ [B_2=C_2]\ ^{L}T$$

This join may be represented as a hierarchy of joins as depicted in Figure 2.

At first glance, to apply the full predicate coverage principle, the whole join may be formulated as a conjunctive expression such as $J_0 \wedge J_1$. Then, two sets of coverage rules may be obtained (one for each join). The first one would ensure that J_0 will have all possible evaluations while maintaining J_1 to true (inner), which can be accomplished using Definition 9 by applying transformations $\Phi_J(I,J_0)$, $\Phi_J(L,J_0)$ and $\Phi_J(R,J_0)$ to J_0 respectively and transformation $\Phi_J(I,J_1)$ to J_1 in all cases. The second one is symmetric. This interpretation is exactly the same as that of select operators, however, because the semantics of join is different some issues related to the nested join have to be considered as shown in this section.

Consider the relations R={(11,x),(12,y),(13,z)}, S={(21,x,t),(22,y,u)} and T={(31,t)}. The join depicted in Figure 2 produces the result displayed in Table 2.

Assume that Δ_L rules are being generated for J_0 and J_1, respectively, while maintaining J_1 and J_0, respectively, to inner. The resulting rules would be:

$$\Delta_L(J_0) := (\ (R[A_1=B_1]^{I}S)\ [B_2=C_2]\ ^{L}T\ )\ [\neg nl(B_2) \wedge nl(C_2)]$$

$$\Delta_L(J_1) := (\ (R[A_1=B_1]^{L}S)\ [B_2=C_2]\ ^{I}T\ )\ [\neg nl(A_1) \wedge nl(B_1)]$$

The first coverage rule $\Delta_L(J_0)$ will work correctly. Its first inner join on R and S generates a relation in which J_1 is evaluated to inner (I), producing tuples of R and S in rows 1 and 2, but not row 3 (Table 2). The resulting relation is left joined with T resulting in a relation in which J_0 is evaluated to I (row 1) and L (row 2). Then the select predicate $[\neg nl(B_2) \wedge nl(C_2)]$ will keep only those tuples that evaluate J_0 to L (row 2).

However the second coverage rule $\Delta_L(J_1)$ will not return any tuple. The first join (left join) on R and S generates a relation in which J_1 is evaluated to L (producing rows 1, 2 and 3). The second one (inner join) will discard rows 2 and 3. Then the select predicate $[\neg nl(A_1) \wedge nl(B_1)]$ will discard row 1. The resulting relation is empty. In this case the correct join type for the rule $\Delta_L(J_1)$ would be L for both J_1 and J_0 which would have produced row 3.

A new definition of Φ_J is needed in order to determine the correct join type for each join. That will be accomplished by means of a function which labels each join with the correct label type.

Definition 12. (Nested join and labelling) Let R[p]S be a join operator, denoted as J. A join operator J is said to be a nested join, if some of the relations R or S are derived relations obtained as the result of another join. Each of the constituent joins is denoted by J_i such that the root is the first element J_0. The *Missing Values Outer Increment* (MVOI) is the set of base relations whose values are filled with missing values because of the evaluation of outer joins in the nested join J.

Let J_i be the test condition which must be evaluated to a join type JT in {L,R}. Let loirels(J_i) and roirels(J_i) be the set of base relations that are filled with missing values when J_i generates the LOI and ROI, respectively. The *Nested Join Labelling* function NJL(JT,J,i) returns an array label[] which determines the correct join type label[k]$\in${L,R,I} for each J_k in J, given a target

join J_i which must be evaluated to JT. The algorithm displayed in Figure 3 details the procedure for evaluating J_i to L (the procedure for R is symmetric).

Consider the process of generating the correct labels for joins in coverage rule $\Delta_L(J_1)$ in the example 3. The labelling proceeds first by selecting the target join (J_1). Next, it assigns label L to this join and adds the relation S to MVOI. The loop proceeds by examining the join J_0 (which is not labelled yet). As it joins relations using a join predicate on the attributes of S and T, roirels(J_0)={S} (which is in MVOI). Then it assigns label L to J_0 and finishes with the correct set of labels {L,L}.

D*efinition 13. (Coverage rules and transformations for a nested join operator)* Let J be a nested join and J_i each of its constituent joins. The nested join type transformation modifies the join type of every constituent join J_k as indicated below:

$$\Phi_{JN}(I,J) := \Phi_J(I,J_k)\ \forall J_k \text{ in } J$$

$$\Phi_{JN}(L,J,J_i) := \Phi_J(\text{label}[k],J_k)\ \forall J_k \text{ in } J, \text{ where label}=NJL(L,J,i)$$

$$\Phi_{JN}(R,J,J_i) := \Phi_J(\text{label}[k],J_k)\ \forall J_k \text{ in } J, \text{ where label}=NJL(R,J,i)$$

The coverage rules Δ_I, Δ_L and Δ_R select the tuples that belong to the inner join, and the left and right outer increments for each J_i, respectively, and are defined as:

$$\Delta_I(J) := \Phi_{JN}(I,J) \quad (9)$$

$$\forall J_i \text{ in } J : \Delta_L(J_i) := \Phi_{JN}(L,J,J_i)\ [\Phi_{LOI}(J_i)] \quad (10)$$

$$\forall J_i \text{ in } J : \Delta_R(J_i) := \Phi_{JN}(R,J,J_i)\ [\Phi_{ROI}(J_i)] \quad (11)$$

Example 4. Consider a nested join J={J_0,J_1,J_2}on four base relations which first evaluates two joins: J_1 on R and S and J_2 on T and U and finally evaluates the root join J_0 using the result of the other relations. Nested join J is depicted in Figure 4 and represented as:

$$Y \leftarrow (R[A_2=B_1]^L S)\ [A_1=D_1]^I\ (T[C_2=D_1]^I U)$$

For instance, when generating the $\Phi_J(L,J_0)$ the function NJL(L,J_0) first labels it with L. Now, labels={L,∅ ,∅} and MVOI={U} because loirels(J_0)={U} (join J_0 adds null values in D1). Next it picks join J_1 but because neither loirels(J_1)={S} nor roirels(J_1)={R} are included in the MVOI, its label does not change. Next it picks join J_2, because loirels(J_1)={U} is included in the MVOI, it labels it with R and adds roirels(J_1)={T} to the MVOI. Now, labels={L,∅,R} and MVOI={U,T}. The algorithm finishes because no more joins can be labelled either to L or R and completes the labels with I. Finally, labels={L,I,R}. The resulting coverage rules are:

$\Delta_I(J) :=$ ($(R[A_2=B_1]^I S)$ $[A_1=D_1]^I$ $(T[C_2=D_1]^I U)$)
$\Delta_L(J_1) :=$ ($(R[A_2=B_1]^L S)$ $[A_1=D_1]^I$ $(T[C_2=D_1]^I U)$) $[\neg nl(A_2) \wedge nl(B_1)]$
$\Delta_R(J_1) :=$ ($(R[A_2=B_1]^R S)$ $[A_1=D_1]^L$ $(T[C_2=D_1]^R U)$) $[nl(A_2) \wedge \neg nl(B_1)]$
$\Delta_L(J_0) :=$ ($(R[A_2=B_1]^I S)$ $[A_1=D_1]^L$ $(T[C_2=D_1]^R U)$) $[\neg nl(A_1) \wedge nl(D_1)]$
$\Delta_R(J_0) :=$ ($(R[A_2=B_1]^L S)$ $[A_1=D_1]^R$ $(T[C_2=D_1]^I U)$) $[nl(A_1) \wedge \neg nl(D_1)]$
$\Delta_L(J_2) :=$ ($(R[A_2=B_1]^L S)$ $[A_1=D_1]^R$ $(T[C_2=D_1]^L U)$) $[\neg nl(C_2) \wedge nl(D_1)]$
$\Delta_R(J_2) :=$ ($(R[A_2=B_1]^I S)$ $[A_1=D_1]^I$ $(T[C_2=D_1]^R U)$) $[nl(C_2) \wedge \neg nl(D_1)]$

4. COVERAGE RULES FOR COMBINATIONS OF OPERATORS

In the previous section the construction of the coverage rules for queries that include either join or select operators has been detailed. However, usual queries consist of a combination of join and select operators. This kind of query joins several relations and then selects some of the tuples of the resulting relation, in the form:

$$Z \leftarrow (R[p(A,B)]S)\ [q(A,B)]$$

The combination of both operators may be informally formulated like a conjunction such as $J \wedge q$ where J represents the conjunction of all test conditions of the (nested) join and q is the select predicate. At first glance, each of the rules for joins presented in Section 3.3 may be generated while maintaining the select predicate to true. Conversely, each of the rules for the select operator presented in Section 3.2 may be generated while maintaining the joins to true (inner join).

For instance the rule Δ_L for the join J would be obtained from formula (5) and adding q to the select predicate:

$$\Delta_L(J) := \Phi_J(L,J)\ [\Phi_{LOI}(J) \wedge q]$$

And the rule Δ_T for test condition $c_i \in bpreds(q)$ would be obtained from formula (1) applied to the resulting relation after transforming the join to inner using formula (4):

$$\Delta_T(q,c_i) := \Phi_J(I,J)\ [\Phi_T(q,c_i)]$$

However, the approach is not so simple due to dependencies between the attributes in join and select operators. The detailed construction of rules for different combinations of operators is presented in this section.

4.1. Rules for join operators with select

Introductory example 4. Consider a join operator (J) which is followed by a select predicate:

$$Z \leftarrow R[A_1=B_1]^F S\ [B_0=0]$$

The coverage rule for the join Δ_L would be obtained from formula (5):

$$\Delta_L(J) := (R[A_1=B_1]^L S)[\neg nl(A_1) \wedge nl(B_1) \wedge B_0=0]$$

However, because of the construction of the relation (left join), all tuples have missing values in S, so, B_0 is always null and the rule does not return any tuple. The predicate in the select operator needs to be transformed in some way in order to avoid this situation. In this case all attributes in the predicate that may be null because of the left/right joins must be reduced by applying a Null Reduction transformation $NR(p,A_k)$ presented in Definition 5. The set of relations that have all their values null as a consequence of the left/right joins is determined using the algorithm presented in Figure 3.

Definition 14. (Missing values outer increment of a join and join type). Let J be a nested join and J_i the test condition, and a join type JT in {L,R}. The Missing values outer increment of J_i with respect to the join type JT, denoted as MVOI(JT,J,i) is the set of relations that are filled with missing values when considering the outer increments generated by the evaluation of J_i to the join type JT. The MVOI(JT,J,i) is calculated according to the algorithm depicted in Figure 3 with the only difference that it returns the computed MVOI set instead of the label array.

Definition 15. (Coverage rules for a join operator with a select). Let J be a nested join and R[q] be a select operator such that R is the result of the join J. Let the *Null Reduction for a Set of relations* NRS(q,S) be a transformation of q with respect to a set of relations S which reduces every attribute $A_k \in attrs(S)$ by applying $NR(q,A_k)$ from Definition 5. The set of coverage rules in Definition 13 is modified to take into account the select operator as follows:

$$\forall J_i \text{ in } J : \Delta_L(J_i) := \Phi_{JN}(L,J,J_i)\ [\Phi_{LOI}(J_i) \wedge NRS(q,MVOI(L,J,i))] \quad (12)$$

$$\forall J_i \text{ in } J : \Delta_R(J_i) := \Phi_{JN}(R,J,J_i)\ [\Phi_{ROI}(J_i) \wedge NRS(q,MVOI(R,J,i))] \quad (13)$$

Note that $\Delta_I(J)$ rule has been removed. This rule is not generated for joins when there is a select predicate because the rules generated for the select predicate include at least a rule that makes the select predicate true and keeps all joins to inner (see next section).

In the above Example 4, the coverage rules are:

$\Delta_L(J) := (R[A_1=B_1]^L S)[\neg nl(A_1) \wedge nl(B_1)]$
$\Delta_R(J) := (R[A_1=B_1]^R S)[\neg nl(B_1) \wedge nl(A_1) \wedge B_0=0]$

Note that the predicate $B_0=0$ has been removed in the first rule Δ_L because it has been reduced, but not in Δ_R.

4.2. Rules for select operators with join

Using the simple approach presented at the beginning of this section, coverage rules for the select operator may be generated as presented in Section 3.2, and then transforming all joins to inner join. In this case, there is no problem with missing values that may appear because the inner joins do not generate any missing value for any relation. However, now the problem is with missing values that may disappear when all joins are transformed to inner.

Introductory example 5. Consider a select predicate which is applied to the result of a join operator. Assume that A_0 is not nullable:

$$Z \leftarrow R[A_1=B_1]^F S\ [A_0=0 \vee nl(A_0)]$$

The query selects all tuples from the inner join and the left outer increment in which A_0 is 0 plus all tuples in the right outer increment (because it allows tuples with null values in A_0). Note that this predicate is correctly stated even if A_0 is not nullable, as the full outer join may produce null values for this attribute in the ROI. When generating the rule $\Delta_T(p, nl(A_0))$ for the select operator, its predicate is transformed into $nl(A_0)$ ($A_0=0$ is reduced) and the join is transformed to inner join:

$$\Delta_T(p, nl(A_0)) := R[A_1=B_1]^I S\ [nl(A_0)]$$

However, because the join has been transformed into inner join, there is no ROI, so, the attribute A_0 can only be null if it is nullable. If A_0 is not nullable the rule does not produce any result.

This situation arises when a non nullable attribute appears under the null check predicate. In this case, although not frequent, in order not to produce a rule which is impossible to be covered, the predicate of the coverage rule must be checked before generating the rule. If the predicate is impossible to be fulfilled (this is accomplished by quantifying the predicate), then the transformation of the joins is relaxed and all joins are kept with their original join type.

The procedure for quantifying the predicate q traverses every base predicate in p. At each one, if a null predicate over non nullable attributes is found, it is replaced by false. The resulting predicate is traversed again in a depth first order. For each sub-predicate, if it is in the form $q \vee false$, it is replaced by q. If the sub-predicate is in the form $q \wedge false$ then the entire sub-predicate is removed. If the result at the end is an empty predicate or false, then the joins are not transformed into inner join.

4.3. Rules for joins and select operators before and after framing

The most complete query is that which combines one or more joins followed by a select operator, then frames and groups the result and lastly applies another select operator to the resulting groups (in SQL, it includes one or more JOIN and WHERE, GROUP BY and HAVING clauses). This kind of query is in the form:

$$Z \leftarrow R[p(A,B)]^{JT} S\ [q(A,B)]\ ///\ G\ [s(G,F)],\ \text{with } G \subseteq A \cup B$$

In SQL (assuming that the join type is left, and R and S are base relations):

SELECT G, F FROM R LEFT JOIN S ON p(A,B) WHERE q(A,B)
GROUP BY G HAVING s(G,F)

Definition *16. (Coverage rules for nested join and select operators before and after framing).* Let J [q] /// G [s] be an RM/V2 operator which joins several relations (denoted by the nested join

J), then selects tuples based on the select predicate q (WHERE clause), then frames and groups the result based on the set of attributes G (GROUP BY clause) and finally selects the resulting tuples based on the select predicate s (HAVING clause). The complete set of coverage rules are defined as indicated below:

A set of join rules as in Definition 15, which transform J and q. The predicate s is also transformed by applying NRS to the MVOI (Definition 14):

$$\forall J_i \text{ in } J: \Delta_L(J_i) := \Phi_{JN}(L,J,J_i)[\Phi_{LOI}(J_i) \wedge NRS(q,MVOI(L,J,i))] \text{ /// } G[NRS(s,MVOI(L,J,i))]] \quad (14)$$

$$\forall J_i \text{ in } J: \Delta_R(J_i) := \Phi_{JN}(R,J,J_i)[\Phi_{ROI}(J_i) \wedge NRS(q,MVOI(R,J,i))] \text{ /// } G[NRS(s,MVOI(R,J,i))]] \quad (15)$$

A set of rules for predicate q (WHERE clause) as in Definition 6 which transform q. The join operators J are transformed using $\Phi_{JN}(I,J)$ (as in Definition 13) and the reduction NR (definition 5) is also applied to s for rules Δ_N:

$$\forall c_i \in bpreds(q): \Delta_T(q,c_i) := \Phi_{JN}(I,J) \ [\Phi_T(q,c_i)] \text{ /// } G \ [s] \quad (16)$$

$$\forall c_i \in bpreds(q): \Delta_F(q,c_i) := \Phi_{JN}(I,J) \ [\Phi_F(q,c_i)] \text{ /// } G \ [s] \quad (17)$$

$$\forall c_i \in bpreds(q), \forall A_k \in battrs(c_i) \mid nullable(A_k): \Delta_N(c_i,A_k) := \Phi_{JN}(I,J) \ [\Phi_N(c_i,A_k)] \text{ /// } G \ [NR(s,A_k)] \quad (18)$$

Symmetrically, a set of rules for predicate s (HAVING clause) as in Definition 6. The join operators are transformed using $\Phi_{JN}(I,J)$ and the reduction NR is also applied to q for rules Δ_N:

$$\forall c_i \in bpreds(s): \Delta_T(s,c_i) := \Phi_{JN}(I,J) \ [q] \text{ /// } G \ [\Phi_T(s,c_i)] \quad (19)$$

$$\forall c_i \in bpreds(s): \Delta_F(s,c_i) := \Phi_{JN}(I,J) \ [q] \text{ /// } G \ [\Phi_F(s,c_i)] \quad (20)$$

$$\forall c_i \in bpreds(s), \forall A_k \in battrs(c_i) \mid nullable(A_k): \Delta_N(c_i,A_k) := \Phi_{JN}(I,J) \ [NR(q,A_k)] \text{ /// } G \ [\Phi_N(c_i,A_k)] \quad (21)$$

5. ADVANCED RULES

Usually, SQL queries may contain other common constructs in addition to selection and joining. For instance, the Transaction Processing Council (TPC) benchmarks such as TPC-H include many queries with unions, subqueries and framed relations (including the evaluation of aggregate functions). In addition, case expressions are a frequently used way to obtain derived attributes from logical expressions.

In this section, the generation of coverage rules for this kind of constructs is presented. Section 5.1 deals with case expressions, Section 5.2 deals with unions and subqueries. Sections 5.3 and 5.4 deal with framed relations and the aggregate functions.

5.1. Case expressions

In SQL, a *case expression* is a function that specifies a conditional value. For example, the following expression returns v_1 if p_1 is true; if not, it returns v_2 if p_2 is true; if not, it returns the value specified by the ELSE clause (which is optional):

CASE WHEN p1 THEN v1 WHEN p2 THEN v2 ELSE v3 END CASE

The case expression may be considered as a derived attribute whose value is determined as a function $f(p_1..p_n)$ of a set of predicates p_i. The approach to obtain the coverage rules for the case expression consists in transforming its predicates p_i into a single predicate and then to generating the coverage rules defined for select operators taking each base attribute of the predicates p_i as the test conditions.

Definition 17. (Coverage rules for a case expression in a select operator) Let R[p] be a select operator. Let $f(q_1..q_n)$ be a derived attribute of p representing a case expression. Let the *Case*

Conditional Predicate $ccp(f)=q_1 \wedge \ldots \wedge q_n$ be the logical conjunction of all predicates in the case expression. The set of coverage rules for f is obtained as in Definition 6, but applied to ccp(f) after appending it to the select predicate p:

$$\forall c_i \in bpreds(ccp(f)): \Delta_T(f,c_i) := R[p \wedge \Phi_T(ccp(f),c_i)] \quad (22)$$

$$\forall c_i \in bpreds(ccp(f)): \Delta_F(f,c_i) := R[p \wedge \Phi_F(ccp(f),c_i)] \quad (23)$$

$$\forall c_i \in bpreds(ccp(f)), \forall A_k \in battrs(c_i) \mid nullable(A_k): \Delta_N(f,c_i,A_k) := R[p \wedge \Phi_N(ccp(f),c_i,A_k)] \quad (24)$$

Coverage rules for the SQL case abbreviations NULLIF and COALESCE are generated in the same way after transforming the case abbreviation into its corresponding case expression as specified in the SQL standard.

Case expressions may appear as an attribute (in the select list or in grouping attributes) or in a select predicate. When a case expression appears in a select predicate, it is removed before generating the rules. If there is a framing the above transformations are applied to the predicate after the framing (HAVING).

5.2. Multiple queries

All coverage rules presented before this section are designed for a single RVE expressing a single query. However, a query may be composed of other queries in the following cases:

- Several RVEs expressing queries are concatenated by the *union* operator (UNION clause). In this case, an independent set of coverage rules is generated for each of the queries that participate in the union.
- An RVE expressing a query appears as a relation in a join operator (in the FROM clause in SQL): These are named *derived tables*. In this case an independent set of coverage rules is generated both for the main query and the derived table.
- An RVE expressing a subquery appears as a derived attribute (when it is included in a *scalar subquery*), or as a logical predicate (when it is included in an IN predicate, an EXISTS predicate or a *quantified comparison predicate* with ALL, SOME or ANY). Subqueries can not be tackled independently from their main query when generating coverage rules as some attributes may be correlated with relations of the main query; therefore they need some context provided by the relations of the main query.

The approach taken for subqueries is similar to the one taken for case expressions. A predicate is to be added to the main query in order to include the requirements imposed by the full predicate coverage criterion.

Definition 18. (Coverage rules for a subquery) Let R[p] be a select operator. Let Q be a subquery and $\Delta_i(Q)$ each of the coverage rules generated for Q when considered isolated from the main query. Let exists(Q) be a boolean predicate which is true iff the relational assignment of Q results in a non empty relation. The set of coverage rules for Q is obtained by embedding each of the coverage rules for Q in the select predicate:

$$\forall \Delta_i(Q) \text{ obtained from Q: } \Delta_{Qi}(Q,p) := R[p \wedge exists(\Delta_i(Q))] \quad (25)$$

Similarly to the case expressions, if the subquery appears in a select predicate, it is removed before generating the rules. If there is a framing the above transformations are applied to the predicate after the framing (HAVING).

5.3. Framed relations

The previous coverage rules have been designed to tackle the most common operators for joining and selecting data from a set of relations. This section deals with the specific coverage rules that

are obtained from framed relations (in SQL, queries that include the GROUP BY clause). As explained in Section 2.2, a framed relation is in the form:

$$Z \leftarrow R \text{ /// } G, \text{ in SQL: SELECT G,F FROM R GROUP BY G}$$

The SQL Standard [1] specifies how each tuple (row) is grouped according to the set of grouping attributes G (grouping columns): "The result of the <group by clause> is a partitioning of the rows of T into the minimum number of groups such that, for each grouping column of each group, no two values of that grouping column are distinct"

Consider, for example, two tuples j, k on attributes A_1, A_2: (A_1^j,A_2^j) and (A_1^k,A_2^k). The predicate that determines whether both tuples belong to the same group is $A_1^j=A_1^k \wedge A_2^j=A_2^k$. Then a set of coverage rules may be created to fulfill the full predicate principle with respect to this predicate.

Definition 19. (Evaluation of a framed relation) Let R(A) /// G be a frame operator, where $G \subseteq A$ is the set of grouping attributes. Let R^j be each tuple j in R, and A_i^j each value of the attribute A_i at the tuple j. A pair of tuples j, k belong to the same group with respect to the grouping attributes $G \subseteq A$ iff $\underset{A_i \in G}{\wedge} A_i^j = A_i^k$. This predicate is named *grouping predicate*.

Consider the previous example. Three coverage rules may be generated to check whether the attributes A_1 and A_2 determine the value of the grouping predicate: (1) $A_1^j=A_1^k \wedge A_2^j=A_2^k$, (2) $A_1^j \neq A_1^k \wedge A_2^j=A_2^k$ and (3) $A_1^j=A_1^k \wedge A_2^j \neq A_2^k$. In order to be consistent with the definitions of coverage rules in previous sections, the corresponding coverage rules must be expressed in terms of three RVEs to check that there exists at least a pair of tuples such that each of the rules (1), (2) and (3) are fulfilled, respectively. Intuitively:

- For rule (1) the original RVE may be transformed into another one which returns only those tuples that have groups composed of at least two tuples.
- For each of the rules (2) and (3) the original RVE may be transformed into another one which excludes each grouping attribute G_1, G_2, respectively, and returns only those tuples that have groups composed of at least two tuples, and distinct values in A_1, A_2, respectively.

Definition 20. (Coverage rules and transformations for framed relations) Let R(A) /// G be a frame operator, where $G \subseteq A$ is the set of grouping attributes and G_i denotes the ith grouping attribute. Let count(*) be a function that returns for each group the number of tuples in the group. Let countd(A_i) be a function that returns for each group the number of tuples that have different values of A_i in the group. The coverage rule transformations to attain full predicate coverage of G are:

$$\Phi_G(R \text{ /// } G) := R \text{ /// } G \text{ [count(*)>1]}$$

$$\Phi_{GA}(R \text{ /// } G , G_i) := R \text{ /// } X \text{ [countd}(G_i)>1], X=G-\{G_i\}$$

Then the set of coverage rules Δ_G and $\Delta_{GA}(G_i)$ are:

$$\Delta_G(R \text{ /// } G) := \Phi_G(R \text{ /// } G) \quad (26)$$

$$\forall G_i \in G,\ \Delta_{GA}(R \text{ /// } G , G_i) := \Phi_{GA}(R \text{ /// } G, G_i) \quad (27)$$

A final remark has to be made related to the second rule Δ_{GA}. If for any relation the grouping attributes contain all its primary keys and some other non key attributes, the coverage rules Δ_{GA} for the non key attributes are impossible to be fulfilled and therefore they are not generated.

Example 5. Consider a framed relation R /// G where $G=\{A_1,A_2\}$. The coverage rules for the framed relation are:

$\Delta_G(R///G) := R /// \{A_1,A_2\}$ [count(*)>1]
$\Delta_{GA}(R///G, A_1) := R /// \{A_2\}$ [countd(A_1)>1]

$\Delta_{GA}(R///G, A_2) := R /// \{A_1\} [countd(A_2)>1]$

5.4. Aggregate functions

Aggregate functions (named *set functions* in the SQL Standard [1]) are functions that perform statistic computations over the values of an attribute for each of the groups. These functions are avg, max, min, sum, count. Each function is qualified by DISTINCT or ALL (if none specified, ALL is assumed by default). Two relevant conditions control the behaviour of this computation: (1) If distinct is specified, redundant duplicate values are eliminated and (2) If there is one or more null values they are eliminated.

Intuitively, two different rules may be generated to test these conditions, which for a given attribute A_i require the existence of a group such that: (1) there exist two tuples with equal values in A_i (for removal of duplicates) and (2) there exists a tuple with a null value in A_i and another one with a non null value in A (for removal of null values). An additional tuple is to be required in order to obtain an evaluation of the aggregate function over at least two values.

Definition 21. (Evaluation of aggregate functions) Let $f(A_i)$ be an aggregate function with the default qualifier ALL {avg, max, min, sum, count} or with the optional qualifier DISTINCT {avgd, maxd, mind, sumd, countd}. Let R^j be each tuple j in a group of a framed relation R///G, and A_i^j each value of the attribute A_i at the tuple j. The test conditions are: (1) $A_i^j=A_i^k$, $j\neq k$ and (2) $nl(A_i^j)$ provided that the resulting group contains at least two tuples. As in previous sections the coverage rules must be expressed as an RVE.

Definition 22. (Coverage rules and transformations for aggregate functions) Let R(A) /// G [q] be a frame operator, where $G\subseteq A$ is the set of grouping attributes G_i. Let $f(A_i)$ be an aggregate function over the attribute A_i The coverage rule transformations to attain full predicate coverage of F with respect to A_i are:

$$\Phi_A(R /// G [q], A_i) := R /// G [q \wedge count(A_i)> countd(A_i) \wedge countd(A_i)>1]$$

$$\Phi_{AN}(R /// G [q], A_i) := R /// G [q \wedge count(*)>count(A_i) \wedge countd(A_i)>1]$$

Then the set of coverage rules $\Delta_A(A_i)$ and $\Delta_{AN}(A_i)$ are:

$$\Delta_A(R /// G, A_i) := \Phi_A(R /// G, A_i) \quad (28)$$

$$\Delta_{AN}(R /// G, A_i) := \Phi_{AN}(R /// G, A_i), \text{ if } A_i \text{ is nullable} \quad (29)$$

Example 6. Consider a framed relation R /// $\{A_1\}$ [sum(A_2)=10]. The coverage rules for the aggregate function sum are:

$\Delta_A(R///G, A_2) := R /// \{A_1\} [sum(A_2)=10 \wedge count(A_2)>countd(A_2) \wedge countd(A_2)>1]$
$\Delta_{AN}(R///G, A_2) := R /// \{A_1\} [sum(A_2)=10 \wedge count(*)>count(A_2) \wedge countd(A_2)>1]$

6. SUMMARY AND TOOL SUPPORT

This section first provides a summary of all coverage rules (Δ) and coverage rule transformations (Φ) that have been defined so far. Table 3 summarizes for each syntactic element and combination in a query (first column), the applicable coverage rules (second column) and the definition in which the rules are described (third column). The applicable transformations to obtain the rule are also summarized (fourth column) along with the definition in which they are described (fifth column). Finally, the lower part of the table summarizes other additional transformations that are used in the definitions.

The generation of the SQLFpc coverage rules described above has been completely implemented in a set of tools, available at http://in2test.lsi.uniovi.es/sqlfpc. The architecture is depicted in Figure 5 and the available tools are described below:

- SQLFpcWeb: A Web interface from which the user can generate the rules interactively. Using a web browser the user specifies the query and information about the database schema (tables, columns, datatypes and other constraints). Each generated rule contains the SQL query that is to be executed for evaluating the coverage and a textual description of the test point requirement that is satisfied if the rule is covered.
- SQLFpcWS: A Web Service that performs the same function. It is intended to be integrated with other applications. The external application has to specify the query and the information about the database schema in an internal XML format and then invoke the service, which returns the rules embedded in an XML document that can be further processed. A complementary tool named XDBSchema may be used to obtain the XML representation of the schema from a local database.
- SQLRules: A standalone Java application that allows both the generation and evaluation of the rules against a previously populated database. The user specifies the query and the information required to connect to a local database. Then the database schema is automatically extracted using the standard java jdbc methods to access the metadata. Views are considered like tables. Finally, the web service (SQLFpcWS) is invoked and the generated rules displayed. The user has an additional option to execute the rules against the existing data. Then the rules that are covered along with the percent coverage are displayed. Also, if the query has parameters, there is the possibility of specifying the actual parameter values that will be used when evaluating the coverage.

Internally, the SQLFpc core is a set of classes that first parse the SQL statement received and store it along with the database schema in internal objects. After that, it proceeds to generate each of the rules by applying the transformations described in previous sections. The database schema is used to determine the constraints (mainly referential and nullability) which condition how the transformations are applied.

In a common scenario, a tester creates one or more test suites, each of which may have its own database. Then the tester executes the test suite and evaluates the coverage of the test database against each executed query. The information may be used to modify the test database if needed to increase the coverage.

All coverage rules described in previous sections handle individual select queries, and all coverage rules described have been implemented in the tools and thoroughly tested. However, there are a number of limitations to its use in database applications that are described below.

The SQLFpc criterion focuses on the queries that retrieve information from the database (SELECT queries), which are the most frequently used SQL statements in commercial applications as shown by Pönighaus [21] that reports that select queries are the 68% of the total number of queries. However, the other SQL main statements that modify the database state are not directly supported. These statements sometimes perform some kind of selection of data and coverage rules for this selection could be generated. For instance, the INSERT statement is composed of several assignments of values to columns along with an optional SELECT clause to select from the database the values for the inserted rows. So as, the coverage rules for this clause could be generated and used to measure the coverage. Similarly, the UPDATE and DELETE statements may use a WHERE clause to select which rows will be updated or deleted. In this case the coverage rules for the where clause could also be used to measure the coverage.

The SQL queries are considered in isolation by the current tools. That means that in order to evaluate the coverage of the statements that are executed from an application, additional work should be done by instrumenting the program and calling the web service (SQLFpcWS) to obtain and execute the rules against the query that is being executed. A tool for doing so would be similar to the one developed by Zhou and Frankl [22] which executes the SQL mutants generated from the queries embedded in Java programs. Nevertheless, the problem of identifying all queries that may be executed by the program is still present, although the most complete is the

coverage of the imperative parts of the program, more queries will be executed and their coverage analyzed.

Finally, the current implementation covers a wide range of usual SQL constructs and has been tested using SQL Server and Oracle database management systems. As DBMS vendors include different features that are sometimes different from the SQL standards, it is possible to have queries that fail to generate the rules because of features that are not yet considered in the supported SQL syntax. A mechanism to report problems and propose enhancements is embedded in the tools.

7. CASE STUDY

In this section, a case study is presented using a set of queries obtained from an open source real-life Enterprise Resource Planning (ERP) application. Firstly, in Section 7.1 the set of queries used and the test databases are described. Next in Section 7.2 the results of the generation and execution of the rules are presented. Finally, in Section 7.3 the process of developing test cases for one of the queries is explained.

7.1. Set of queries and test database

Compiere is an open source ERP and Customer Relationship Management (CRM) business solution for Small and Medium-sized Enterprises. In this case study a set of queries taken from this application is used. In particular, the queries constitute the full set of views[4] that are packaged with the application. From the testing point of view, this set is one of the most interesting as the views are intensively used by the rest of the queries of the application and because they contain complex queries.

This set of queries combines small and large queries which use many different tables in the database. Appendix A contains a listing of each query along with their main characteristics measured in terms of number of SELECT clauses, conditions in WHERE and HAVING, tables and case expressions. In total, there are 107 queries. The largest queries have up to 24 conditions in the WHERE clause (query named C_Invoice_Candidate_v), joins of up to 15 different tables (C_Order_Header_v) and up to 19 case expressions (C_Dunning_Line_vt). Some queries are composed of the union of many queries: the largest ones include 15 queries (RV_UnPosted), and 5 queries with joins over 22 tables (C_Invoice_LineTax_vt). The set of queries uses a total of 136 different tables from the Compiere database schema. The tables have an average of 23 columns; C_PAYMENT is the largest with 84 columns.

Before the generation and execution of the rules, a database populated with data must be obtained. The approach taken was generating four different test databases with different sizes, in order to show the performance of the rules when executing against the databases. The open source tool dbMonster[5] was used for such purpose. The resulting databases have 4, 10, 100 and 1000 rows per table under the Oracle XE 10g database management system.

7.2. Generating and running the rules

For each query, the set of coverage rules has been generated using the SQLFpc tools and then executed against each test database in order to calculate the percent coverage. Table 4 depicts a summary of the results. Rows represent each type of rule. The third column shows the number of rules generated, and the last columns the coverage for each of the test databases (sizes 4 to 1000). The details for each query are included in Appendix A.

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

[5] dbMonster is an open source tool that generates random data in a database, available at http://dbmonster.kernelpanic.pl/

In order to assess the fault detection ability of the test data and compare it with the coverage, a set of mutants has been generated for each query using the SQLMutation tool [23] and executed against the test databases. The mutants for SQL have been previously described by the authors [24] and are organized into four categories. The first two categories are specific to the SQL language and the others are similar to those used in procedural code:

- SC - SQL clause mutation operators: These perform mutations on the main clauses: SELECT, JOIN, sub-queries, GROUP BY, UNION, ORDER BY and aggregate functions.
- NL - NULL mutation operators: Mutations related to the handling of null values, whose aim is to ensure that test cases exist that exercise the nulls both in the conditions and the query outputs.
- OR - Operator replacement mutation operators: These include mutations of logical and relational operators and expressions.
- IR - Identifier replacement mutation operators: Replacement of columns, constants and query parameters that are present either in the query or in the tables used by the query.

Table 5 displays a summary of the total number of mutants and scores for each mutant category. The structure is similar to that of the previous table. Details for each query are also included in Appendix A.

The first consideration is made about the number of the rules. The total number of coverage rules is 1,902 (a mean of 17.7 rules per query), which is small compared with the number of mutants: 192,203 (a mean of 1,796 per query). It must be noted, however that the largest mutant category is that corresponding to IR operators, because there are many tables with many columns, and many column references in the queries. If IR operators are not considered, the mean of mutants is 279.5 per query.

As the database increases in size, the coverage attained augments due to the effect of the additional rows. However, on average, the maximum is 51.0% (database with 1,000 rows per table), because many rules are not covered due to complex situations that are not achieved using a random database load. The mutation score shows a similar trend, although starting from upper values. A consequence is that as coverage augments because there is a more complete test input, the mutation coverage also increases. However, it is difficult to increase the coverage and mutation score if the database is randomly generated.

The mutation scores have been calculated as the total number of dead mutants divided by the total number of mutants, excluding those equivalent mutants that are automatically detected. In this experiment the number of equivalent mutants that have been automatically detected for the SC category is 632 which led to a 14% of the SC mutants (there are only 7 equivalent mutants automatically detected in other categories). Another study performed by the authors [24] led to a 6% of equivalent mutants (2.5% were automatically detected), and considering only the SC mutants the percentage was 28% (17% automatically detected). However, in this experiment the version of SQLMutation is able to detect more equivalent mutants and the queries are much more complex than in the former which makes it difficult to estimate the number of equivalent mutants.

A very important practical consideration is related to the performance both for generation and evaluation of the rules. The coverage rules can be used in a number of scenarios: to evaluate the completeness of a given test database in relation to a query, to assist the development of new test cases, to complete a previous test database, or even to assist both the development and testing of a query. In all cases the time spent on the evaluation of the rules is a very important factor, which becomes critical if test cases are being developed or completed in an interactive way. Also, the generation time is important if the query is being developed and tested at the same time.

In Appendix A the times spent on the generation and evaluation of rules for each query are detailed. The generation of rules is made in a single CPU server Xeon 3 Ghz, and execution is made in a desktop P IV 3 Ghz with a local Oracle database server. In summary, the generation time of the full set of rules for all the 107 queries is 45.9 seconds, and the evaluation time ranges from 128.5 seconds (database with 4 rows per table) to 170.5 seconds (database with one thousand rows per table). These numbers show a good scalability on the database size that allows an interactive evaluation of the coverage even with large databases and queries. Considering the individual queries, the average generation time is 0.43 seconds per query and 0.024 seconds per rule. The average evaluation time using the largest database is 1.59 seconds per query and 0.089 seconds per rule. The queries with largest evaluation times (using the largest database) are C_Invoice_Candidate (which spends 11.6 seconds to run 64 rules) and C_Order_Header_v (which spends 9.2 seconds to run 34 rules over 15 tables). These numbers are also an indicator that allows the use of the coverage criterion in an interactive way to develop or complete a test database.

7.3. Constructing test cases using the rules

Some of the queries used in this case study have obtained low coverage (under 50%) due to the fact that they are complex and it is unfeasible to achieve a good coverage using a randomly selected test database. In this section the scenario corresponding to a tester who develops test cases to maximize the coverage is presented. Firstly the test databases for eight queries are developed and the results of coverage and mutation score compared with the random databases. Next a discussion about the coupled conditions detected in the rules is presented. Lastly, the evolution of the coverage and mutation score when adding rows to the test databases and a discussion of the threats to validity are presented.

7.3.1. Generated test databases

The results obtained with the test databases generated for eight of the queries listed in Appendix A are displayed in Table 6. The left part of the table characterizes each query in terms of the number of case expressions (CE), conditions (NC), queries (NQ) and tables (NT). As an example, the SQL of the first query (C_Invoice_Candidate_v) is presented in Appendix B. This query performs a join of four tables plus another one inside a subquery, it has a GROUP BY clause and a complex WHERE clause composed of 24 conditions.

The test input for each query consists of a single database load which is generated in several steps. The process begins with a blank test database and the generation of the coverage rules. At each step the tester adds a test point (a number of rows) to the test database in order to cover a single coverage rule and checks whether it is covered. If not he/she modifies the test point until covered. The middle columns of Table 6 display the number of test points for which any row has been added (TP), the total number of rows that have been added (RW) along with the SQLFpc number of rules (NR) and coverage (Cov.) and the number of mutants (NM) and mutation score (Mut.). The right columns summarize the results of SQLFpc coverage and mutation score presented in Appendix A that have been obtained with random databases of sizes 100 and 1000 rows per table respectively.

A first comparison between the SQLFpc coverage obtained for the developed test databases and that obtained for the random databases with 1000 rows per table shows that the former is always higher than the latter, because the test database has been developed in order to maximize the SQLFpc coverage. The mutation scores are higher in five queries and lower in three (with a small maximum difference of 4.27%). If compared with the random database with 100 rows per table, only two queries achieve lower mutation score (the maximum difference is 2.93%). If comparing with random databases with 10 rows per table (which means a total of 60, 70 and 40 rows for each of the three queries respectively) the mutation scores are always considerably higher for the developed database.

7.3.2. Coupled conditions

Although the SQLFpc coverage is high, it is not 100% in four queries, which means that there are some impossible test points. This is due to the presence of some coupled conditions in WHERE and JOIN clauses. Three different kinds of coupling have been detected:

Coupling in WHERE conditions: Consider the query C_invoice_candidate_v which includes a condition in the form p=(o.ir='S' ∧ nl(bp.isid)) ∨ (o.ir='S' ∧ ¬nl(bp.isid) ∧ other), where other denotes a complex condition and the names of attributes have been simplified for readability. When generating the rule Δ_F(p,¬nl(bp.isid)), after reducing nl(bp.isid) the resulting rule has the following predicate: nl(bp.isid) ∧ (o.ir='S') ∧¬(o.ir='S') ∧ other, which requires the condition o.ir='S' to be true and false simultaneously making the rule impossible to be fulfilled. This kind of coupling could be avoided by putting condition o.ir='S' outside of the parenthesis.

Coupling in WHERE conditions caused by the joins: Consider another part of the same query as above and the join of the relations bp and si: bp[bp.isid=si.isid]^{L}si [¬nl(bp.isid) ∧ (nl(si.if) ∨ si.if=[illegible]]. Note that the attribute si.if is not nullable, but it is checked for nullability because it may belong in the left outer increment, and so, the query is correct. When generating the rule Δ_T(p,nl(si.if)) the resulting rule is bp[bp.isid=si.isid]^{L}si [nl(si.if) ∧ ¬nl(bp.isid)]. Condition si.if=[illegible] has been reduced. However the rule requires ¬nl(bp.isid) which implies that no tuples are generated in the left outer increment, and then si.if can not be null. In this case the condition nl(si.if) in the original query is redundant.

Coupling in JOIN conditions: Consider the query C_Invoice_LineTax_v. A fragment of this query joins relations il, uom, p and pt:

il [il.uomid=uom.uomid]^I uom [il.pid=p.pid]^L p [il.pid=pt.pid ∧ uom.adl = pt.adl]^L pt

As the attribute il.pid is nullable a Δ_{NL} rule is generated for the second join:

il [il.uomid=uom.uomid]^I uom [il.pid=p.pid]^L p [il.pid=pt.pid ∧ uom.adl = pt.adl]^I pt [nl(p.pid) ∧ nl(il.pid)].

Because it requires nl(p.pid) the join condition of the last join evaluates to null and then the rule is impossible. The coupling appears because the same nullable attribute appears in more than one join, which is not frequent, but in this query it causes most of the impossible rules.

7.3.3. Coverage and mutation score trends

For each step the number of rows added to the database, the full predicate coverage and the mutation score has been measured. Figure 6 depicts for each query the evolution of the number of rows added to the database (in percent), coverage and mutation score at each step.

The trend shown by the coverage is nearly linear with the increasing of the size of the database beginning near zero for the first step (because each test point is designed to cover an individual rule). The mutation score begins at a high value (around 40% in most of cases) corresponding to a set of very easy to be killed mutants but also shows an increasing trend as the coverage increases. Queries that have a UNION clause (C_Invoice_Line_Tax_vt, RV_BPartnerOpen and RV_OpenItem) experiment a large increase of mutation score at a middle step, which corresponds with the first step in which test data is developed for the second query of the union. This increasing of mutation coverage is due to the same reason as the high values achieved in the first step of all queries.

Some queries (C_RFQResponseLine_v, C_RFQResponseLine_vt and RV_WharehousePrice) show a small range of variation in the mutation score between the first and last step. The reason is that in these queries, the score for the IR category has a short range of variation and because of the number of IR mutants is much higher than the others, the global effect is also a short range of variation. However, considering only the categories of mutants specifically related to SQL (SC

and NL) the range of variation is considerably larger (16.0% to 76.0%, 10.7% to 71.4% and 5.6% to 44.5%, respectively).

7.3.4. Threats to validity

The above results are an indication that developing test cases for increasing the coverage will also increase the mutation score and therefore the fault detection ability of the test cases (if it is assumed that the mutants are a good indication of the faults that may be present in the query). On the other hand, designing or completing the test database for covering the rules is not a difficult task if the tester has knowledge about the database schema of the tables involved in the query. It is an incremental process in which at each step the tester focuses on covering a single rule and then develops a test point which consists of adding a few rows to the test database (a mean of 1.63 rows per rule considering the eight analyzed queries). However, there are several issues that may threaten the validity of these results, which are discussed below:

Construct validity deals with the issue of whether the measured variables adequately capture the concepts that they are supposed to measure. The mutation score is used to evaluate the fault-detection ability of the test data, but their representativeness of real-life faults may be limited. Previous studies on mutation testing conducted by Andrews et al. [25,26] show that mutation analysis is an appropriate method for evaluating the fault detection capabilities of a test suite. In this experiment, some of the mutants for SQL are similar to the mutants used in the aforementioned studies (OR and IR categories), which contributes to mitigate this threat. However, it should not be forgotten that testing SQL queries is somewhat different to testing imperative programs because of the high input space of test cases and also because a single query can be considered as a small program that performs many complex operations. Furthermore, other additional mutation operators like the one proposed by Kaminski and Amman [27] are able to detect more faults with a lower number of mutants. Also, the mutants used in this study are first order mutants (only one fault is introduced by each mutant); however, some carefully selected high order mutants may be more effective than first order mutants as shown by Jia and Harman [28].

Other mutants are very different than mutants for imperative programs (SC and NL categories). Studies in the errors that humans commit when writing SQL queries (Chan and Wei [29], Brass and Goldberg [30]) conclude that potential problems are spread across all main SQL clauses. The SQL mutants have been constructed to take into account the faults introduced by this kind of errors, which contributes to mitigate this risk. Nevertheless, further research should be done in order to evaluate whether these mutants appropriately model such kind of faults.

Internal validity is concerned with the causal relationship between independent and dependent variables. The test databases have been manually created following the order in which the coverage rules are presented. This could lead to different databases and then different results if they are created by a different tester or by following a different order.

External validity deals with issues that limit the ability to generalize the results. In the experiments the queries are taken from a real application, which contributes to mitigate this threat. However only eight queries have been selected from those that are more complex and all queries are taken from the same application, which may limit the generalization of the results.

8. RELATED WORK

The problem of testing database applications has been addressed in the literature from different perspectives. On some occasions the research is targeted towards the definition of adequacy criteria such as in this paper. On others the focus is on the automatic generation of test data, the checking of the SQL queries sent to the database management system or regression testing. The basic artifact to be tested is on some occasions, the query itself in relation with the test data, on others the program that creates and issues the database queries. In general, in this work,

compared with others, the scope is narrower as it focuses on the evaluation of the coverage of individual SELECT queries, but more complete with regards to the treatment of many details of the SQL language and a more exhaustive measurement of the coverage. In this section a review of related literature is made and the possible use of SQLFpc in the context of the other approaches is also discussed.

Table 7 summarizes the related work. It is organised from the point of view of the goal (assess adequacy, generate test data or other regression testing related activities such as reorder and select test cases or reduce test data) and the criterion that is being used to assess the adequacy or to guide the test data generation. Using a traditional classification of coverage criteria this classification is made with regards to the kind of coverage: structural (control-flow based or data flow based), fault-based or those which use some kind of specification. The table is completed with two additional columns. The scope column indicates whether only SELECT queries are used (select) or another kind of queries like the ones that modify the database are also considered (all). The integration column indicates whether the approach considers single queries (isolated) or queries embedded in an application (embedded).

In the control-based category, a previous work by Suárez-Cabal and Tuya [31,32] incorporates a notion of multiple condition coverage for SELECT and JOIN in individual queries by creating a set of coverage trees. Each of them has a number of coverage nodes that represent the situations to be covered by the query when executed against the test data. A further controlled experiment [5] in which each of the situations to be covered is presented to the user in the form of a textual rule reveals that using this approach leads to test inputs able to reveal more faults in the queries than when it is not used. However, the main drawback is the scalability and performance because the coverage trees may grow exponentially and the evaluation of the coverage is done algorithmically. These issues are overcome by SQLFpc because there is no such exponential growing and the evaluation is efficiently done by the DBMS itself.

Another related approach is that of Chan and Cheung [33], that transforms the SQL query into a procedural language (C++) that can be further used to apply other criteria over the transformed program. This approach may have some of the aforementioned drawbacks, and also the difficulty of generating a translation that preserves the semantics of the queries for a large subset of the SQL language.

A very different, as well as complementary approach is the "command form coverage" proposed by Halfond and Orso [34] that focuses on how the SQL strings containing the query to be executed are constructed by the program. This approach is applicable in many applications in which different variants of a query are elaborated dynamically and sent back to the DBMS in a single interaction point. The coverage is measured in terms of the percentage of different queries that are actually generated in relation to the total number of queries that could be generated. However, neither the coverage of each of the queries nor the test database is taken into account. Therefore, this may be complementary to SQLFpc by using a composite coverage criteria based on the different queries that are exercised and the coverage of each one when executed against the test data.

In the fault-based category the authors developed a set of mutants and a tool for individual SQL queries [23,24]. This has been extended by Zhou and Frankl [22] by means of a tool for executing the mutants of the queries that are embedded in applications written in Java. Chan et al. [35] propose a set of mutants that take into account a database conceptual model and Shahriar and Zulkernine [36] a set of mutants for SQL statements and database method API calls.

In the data-flow based category, Kapfhammer and Soffa [37] define the first set of data-flow criteria based on the definition and use of database entities at different levels of granularity (database, relation, attribute, record and attribute values). The data-flow criteria has been extended by Willmor and Embury [38] and used by Leitao Jr. et al. [39] in the context of active databases. The initial coverage criteria developed by Kapfhammer and Soffa leads to the concept

of "test coverage monitoring" [40] which expands their previous work. The automation of the approach is provided by a tool which instruments the program and the test suite and produces the coverage results. This goal is close to that presented in this paper, which intends to provide a complete analysis tool. However, the criteria are rather different and are not directly comparable. SQLFpc deals with the use of database entities at the record level and its goal is to exercise as much as possible complex queries in relation to the input data. It considers both the data entities that are used and those that are not used, which are also very relevant in testing, because faults in the query may produce both the presence of unexpected outputs and the absence of expected outputs.

In the specification-based category there is a number of very different works. The work of Chays et al. [41,42] in the AGENDA tool which is modified by Deng et al. [43] to include transactions is loosely related to the category-partition method. The user specifies interesting groups (categories) of values. Along with other heuristics and information about constraints of the database schema the groups are combined by a data generator. This produces a set of "test templates" that, after replacing the groups by real values, are used to fill the database. However, this approach on some occasions leads to empty outputs because the different combinations of rows may be filtered due to the presence of other clauses like joins. This problem is further addressed [44] by introducing "generation queries" that select among the combination of rows those that produce some result and are then candidates to be inserted in the database. This procedure may cover some of the coverage rules generated using SQLFpc, but, when queries are more complex, a possible use of the coverage rules as generation queries may produce many other test templates that allow rows to be inserted to exercise the logic of the query more deeply.

A quite different approach is that of Willmor and Embury [45,46] in which the database state required for testing is specified by the tester using intensional constraints in an SQL like language. In this case, the intensional constraints are the criterion used to fill or augment the database. The "reverse query processing" by Binnig et al. [47] uses the desired output (an extensional specification) to generate the test inputs. A further work [48] allows the user to specify the outputs in the form of SQL rules (which is an intensional specification). Both approaches may be able to determine test data that cover the specification and so, may be able to detect different kinds of faults with test data intended to cover the implementation. However, writing the specifications is costly, and if the specification is not fully complete, using just the minimum set of data that covers that specification may lead to other undetected kinds of faults. As in the previous case, the rules generated by SQLFpc criterion (that are also SQL queries) may be used as a complement to the user specified constraints for generating more suitable test data.

Other approaches directed towards test data generation use different kinds of constraint solvers to populate the database. Tsai et al. [49] translate the query into a set of systems of linear inequalities. Zang et al. [50] and Khalek et al. [51] generate a set of constraints that feed a general purpose constraint solver. Emmi et al. [52] use a "concolic execution" that creates a set of constraints intended to achieve coverage of both the source of a Java program and the predicates in WHERE clauses. Starting from an initial database and random inputs the program is executed and some constraints over the not yet covered situations are generated. By solving these constraints more inputs are added to the test and the process is repeated. In these works the adequacy criterion used to guide the generation of the test database is not specifically tailored for databases but predicates or user specified constraints are used. If the SQLFpc coverage rules are translated into the constraints accepted by the solver or directly generated in this language, they may be used to feed the solver to generate more meaningful test data.

Finally, with a different goal, there exists some work on regression testing for database applications that focuses on the order of execution and the number of resets of the test database (Haftmann et al. [53,54]) or the selection of test cases (Haraty et al. [55], and Willmor and Embury [56]). When testing database applications another approach to facilitate regression testing is that of reducing the size of the test database which may lead to test cases that are faster

in relation to the execution and the reset of the database to its initial state, and make the checking of the actual results easier. The SQLFpc coverage rules have been used by the authors with this goal [57]. As the SQLFpc coverage rules retrieve a set of output rows that satisfy each test point requirement, the source tables that produce these rows are traced back and then a small subset of rows kept in the database while the others are discarded. This leads to considerable reductions if the database is large.

9. CONCLUSIONS AND FUTURE WORK

This paper elaborates a coverage criterion (SQLFpc) to assess the adequacy of the data in a test database in relation to the SQL query that is executed. The coverage is based on the well-known MCDC and Full Predicate Coverage criteria. The criterion defines a number of test point requirements for the query under test that lead to executable coverage rules obtained by applying a set of transformations to the query.

The coverage criterion considers a large subset of the SQL language (selection, joining, grouping, aggregate functions, subqueries, case expressions and null values) applicable to SELECT clauses. The elaboration of the test point requirements for the different SQL clauses is detailed, taking into account their specific semantics, as is also the elaboration of the coverage rules for each SQL operator and their combinations.

As a test database becomes more complete, measured in terms of the SQLFpc coverage, the mutation score also increases, making test databases designed for higher coverage to attain better fault detection ability of the queries. Moreover, this approach is fully automated, and the generation and evaluation of the coverage rules against a live database is very efficient. A set of tools is available for such purpose. The tools can be used as an assistant by the tester or developer in evaluating the coverage of test databases, designing test databases from scratch, completing test cases and the simultaneous development and testing of complex queries.

This work may be extended in several directions. A first area of research is the experimentation on how the criteria may help the tester to develop more effective test cases and the trade-offs between cost and quality of the tests, including studies about the fault detection effectiveness. A second area is related to other different applications or tools: In order to allow the evaluation of the coverage for the queries that are issued from an application, the tool support may be improved as discussed in Section 6. Additionally, the coverage rules may be used as a criterion to guide the automatic generation of test databases and to perform a reduction of the database as indicated in Section 8, which is now an ongoing work.

REFERENCES

1. International Standards Organisation. *ISO/IEC 9075, Information technology - Database languages ま SQL*, 1999.
2. McClure RA, Krüger IH. SQL DOM: Compile time checking of dynamic SQL statements. *Proceedings of the 27th International Conference on Software Engineering*. ACM: New York, NY, 2005; 88-96.
3. Waraporn N, Porkaew K. Null semantics for subqueries and atomic predicates. *IAENG International Journal of Computer Science* 2008; 35 (3): 08.
4. Kaminski G, Williams G, Ammann P. Reconciling perspectives of software logic testing. *Software Testing, Verification and Reliability* 2008; 18 (3): 149-188.
5. Tuya J, Dolado J, Suárez-Cabal MJ, de la Riva C. A controlled experiment on white-box database testing. *ACM SIGSOFT Software Engineering Notes* 2008; 33 (1): 1-6.
6. Tuya J, Suárez-Cabal MJ, de la Riva C. A practical guide to SQL white-box testing. *ACM SIGPLAN Notices* 2006; 41 (4): 34-61.
7. Offutt J, Liu S, Abdurazik A, Ammann P. Generating test data from state-based specifications. *Software Testing, Verification and Reliability* 2003; 13(1): 25-53.

8. Chilenski JJ. 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.
9. Codd EF. A relational model of data for large shared data banks. *Communications of the ACM* 1970; 13 (6): 377-387.
10. Codd EF. *The Relational Model for Database Management - Version 2*. Addison-Wesley, 1990.
11. RTCA Inc. *DO-178-B: Software Considerations in Airborne Systems and Equipment Certification.* Radio Technical Commission for Aeronautics (RTCA), 1992.
12. Kapoor J, Bowen JP. Experimental evaluation of the tolerance for control-flow test criteria. *Software Testing, Verification and Reliability* 2004; 14 (3): 167-187.
13. Yu TK, Lau MF. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. *Journal of Systems and Software* 2005; 79 (5): 577-590.
14. Chilenski JJ. Miller SP. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal* 1994; 9 (5): 193-229.
15. Kapoor J, Bowen JP. A formal analysis of MCDC and RCDC test criteria. *Software Testing, Verification and Reliability* 2005; 15 (1): 21-40.
16. Woodward MR, Hennell MA. On the relationship between two control-flow coverage criteria: all JJ-paths and MCDC. *Information and Software Technology* 2006; 48 (7): 433-440.
17. Rajan A, Whalen MW, Heimdahl MPE. The effect of program and model structure on MC/DC test adequacy coverage. *Proceedings of the 30th International Conference on Software Engineering*. ACM: New York, NY, 2008; 161-170.
18. Jones JA, Harrold MJ. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering* 2003; 29 (3): 195-209.
19. Ammann P, Offutt J, Huang H. Coverage Criteria for Logical Expressions. *Proceedings of the 14th International Symposium on Software Reliability Engineering*. IEEE Computer Society: Los Alamitos, CA, 2003; 99-107.
20. Ammann P, Offutt J. *Introduction to Software Testing*. Cambridge University Press: Cambridge, UK, 2008.
21. Pönighaus R. ₹avourite♥SQL-statements - an empirical analysis of SQL-usage in commercial applications. *Proceedings of the 6th International Conference on Information Systems and Management of Data* (Lecture Notes in Computer Science, vol. 1006), Springer, 1995; 75-91.
22. Zhou C, Frankl P, Mutation testing for java database applications. Proceedings of the 2nd International Conference on Software Testing Verification and Validation. IEEE Computer Society, Washington DC, 2009; 396-405.
23. Tuya J, Suárez-Cabal MJ, de la Riva C. SQLMutation: a tool to generate mutants of SQL database queries. *Proceedings of the Second Workshop on Mutation Analysis*. IEEE Computer Society: Los Alamitos, CA, 2006.
24. Tuya J, Suárez-Cabal MJ, de la Riva C. Mutating database queries. *Information and Software Technology* 2007; 49 (4): 398-417.
25. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments?. *Proceedings of the 27th International Conference on Software Engineering*. ACM: New York, NY, 2005; 402-411.
26. Andrews JH, Briand LC, Labiche Y, Namin AS. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 2006; 32 (8): 608-624.
27. Kaminski G, Ammann P. Using a fault hierarchy to improve the efficiencіy of DNF logic mutation testing. *Proceedings of the 2nd International Conference on Software Testing, Verification and Validation*. IEEE Computer Society: Washington DC, 2009: 386-395.
28. Jia Y, Harman M. Constructing subtle faults using high order mutation testing. *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, 2008; 249-258.
29. Chan HC, Wei KK. User□database interface: the effect of abstraction levels on query performance. *MIS Quarterly* 1993; 17 (4): 441□64.
30. Brass S, Goldberg C. Semantic errors in SQL queries: a quite complete list. *Journal of Systems and Software* 2006; 79 (5): 630□44.

31. Suárez-Cabal MJ, Tuya J. Using an SQL coverage measurement for testing database applications. *Proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM: New York, NY, 2004; 253 ま262.
32. Suárez-Cabal MJ, Tuya J. Structural coverage criteria for testing SQL queries. *Journal of Universal Computer Science* 2009; 15(3): 584-619.
33. Chan M, Cheung S. Testing database applications with SQL semantics. *Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*. Wollongong, Australia, 1999; 363-374.
34. Halfond WGJ, Orso A. Command-form coverage for testing database applications. *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society: Washington, DC, 2006; 69-80.
35. Chan WK, Cheung SC, Tse TH. Fault-based testing of database application programs with conceptual data model. *Proceedings of the 5th International Conference on Quality Software*. IEEE Computer Society: Los Alamitos, CA, 2005; 187-196.
36. Shahriar H, Zulkernine M. MUSIC: Mutation-based SQL injection vulnerability checking. *Proceedings of the 8th International Conference on Quality Software*. IEEE Computer Society: Washington DC, 2008: 77ま6.
37. Kapfhammer GM, Soffa ML. A family of test adequacy criteria for database-driven applications. *Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM: New York, NY, 2003; 98ま07.
38. Willmor D, Embury SM. Exploring test adequacy for database systems. *Proceedings of the 3rd UK Software Testing Research Workshop*. York, UK, 2005; 123-133.
39. Leitao Jr OS, Vilela PRS, Jino M. Data flow testing of SQL-based active database applications. *Proceedings of the 3rd International Conference on Software Engineering Advances*. IEEE Computer Society: Washington, DC, 2008; 230-236.
40. Kapfhammer GM, Soffa ML. Database-aware test coverage monitoring. *Proceedings of the 1st Conference on India Software Engineering Conference*. ACM: New York, NY, 2008; 77-86.
41. Chays D, Dan S, Frankl PG, Vokolos FI, Weyuker EJ. A framework for testing database applications. *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM: New York, NY, USA, 2000; 147-157.
42. Chays D, Deng Y, Frankl PG, Dan S, Vokolos FI, Weyuker EJ. An AGENDA for testing relational database applications. *Software Testing, Verification and Reliability* 2004; 14 (1): 17-44.
43. Deng Y, Frankl P, Chays D. Testing database transactions with AGENDA. *Proceedings of the 27th International Conference on Software Engineering*. ACM: New York, NY, USA, 2005; 78-87.
44. Chays D, Shahid J, Frankl PG. Query-based test generation for database applications. *Proceedings of the 1st International Workshop on Testing Database Systems*. ACM: New York, NY, 2008; 1-6.
45. Willmor D, Embury SM. An intensional approach to the specification of test cases for database applications. *Proceedings of the 28th International Conference on Software Engineering*. ACM: New York, NY, USA, 2006; 102-111.
46. Willmor D, Embury SM. Testing the implementation of business rules using intensional database tests. *Proceedings of the Testing: Academic & Industrial Conference on Practice and Research Techniques*. IEEE Computer Society: Washington, DC, 2006; 115-126.
47. Binnig C, Kossmann D, Lo E. Reverse query processing. *Proceedings of the 23rd International Conference on Data Engineering*. IEEE Computer Society: Washington, DC, 2007; 506-515.
48. Binnig C, Kossmann D, Lo E. MultiRQP - Generating test databases for the functional testing of OLTP applications. *Proceedings of the 1st International Workshop on Testing Database Systems*. ACM: New York, NY, USA, 2008.
49. Tsai WT, Volovik D. Keefe TF. Automated test case generation for programs specified by relational algebra queries. *IEEE Transactions on Software Engineering* 1990; 16 (3): 316-324.
50. Zhang J, Xu C, Cheung SC. Automatic generation of database instances for white-box testing. *Proceedings of the 25th International Computer Software and Applications Conference*. IEEE Computer Society: Washington, DC, 2001; 161-165.
51. Khalek SA, Elkarablieh B, Laleye YO, Khurshid A. Query-aware test Generation using a relational constraint solver. *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008; 238-247.

52. Emmi M, Majumdar R, Sen K. Dynamic Test input generation of database applications. *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ACM: New York, NY, 2007; 151-162.
53. Haftmann F, Kossmann D, Kreutz A. Efficient regression tests for database applications. *Proceedings of the 2nd Conference on Innovative Data Systems Research*, Asilomar, CA, 2005: 95-106.
54. Haftmann F, Kossmann D, Lo E. A framework for efficient regression tests on database applications. *The VLDB Journal* 2007; 16 (1): 145-164.
55. Haraty RA, Mansour N, Daou B. Regression test selection for database applications. *Advanced Topics in Database Research, vol. 3*. Siau K (ed). Idea Group Publishing, 2004; 141-165.
56. Willmor D, Embury SM. A safe regression test selection technique for database-driven applications. *Proceedings of the 21st IEEE International Conference on Software Maintenance*. IEEE Computer Society: Washington, DC, 2005; 421-430.
57. Tuya J, Suárez-Cabal MJ, de la Riva C. Query-aware shrinking test databases. *Proceedings of the2nd International Workshop on Testing Database Systems*. ACM: New York, NY, 2009.

TABLES

Table 1. Example of the evaluation of a full outer join

	R		S	
	A_0	A_1	B_0	B_1
Inner join	11	x	21	x
LOI	12	y	null	null
ROI	null	null	22	z

Table 2. Example of a nested join

	R		S			T	
	A_0	A_1	B_0	B_1	B_2	C_2	C_0
(1)	11	x	21	x	t	t	31
(2)	12	y	22	y	u	null	null
(3)	13	z	null	null	null	null	null

Table 3. Summary of all coverage rules and coverage rule transformations

Coverage rules	**Sect.**	**Symbol**	**Def.**	**Applicable transf.**	**Def.**
Single Select	3.2	(1) Δ_T, (2) Δ_F, (3) Δ_N	*6*	Φ_T, Φ_F, Φ_N, IP, NR	*4*, *5*
Select with boundaries	3.2	Δ_{B+}, $\Delta_{B=}$, Δ_{B-}	*7*	Φ_{B+}, $\Phi_{B=}$, Φ_{B-}	*7*
Single Join	3.3	(4) Δ_I, (5) Δ_L, (6) Δ_R	*10*	Φ_J, Φ_{LOI}, Φ_{ROI}	*9*
Single Join (nullable)	3.3	(7) Δ_{NL}, (8) Δ_{NR}	*11*	Φ_{NLOI}, Φ_{NROI}	*11*
Single Nested Join	3.4	(9) Δ_I, (10) Δ_L, (11) Δ_R	*13*	NJL, Φ_{JN}	*12*
Nested Join + Select	4.1	(12) Δ_L, (13) Δ_R	*15*	MVOI, NRS	*14*, *15*
Select + Nested Join	4.2			Transf. for select	
Nested Join + Select + Framing	4.3	(14) Δ_L, (15) Δ_R (16) Δ_T, (17) Δ_F, (18) Δ_N (19) Δ_T, (20) Δ_F, (21) Δ_N	*16*	Transf. for select and join	
Case Expressions	5.1	(22) Δ_T, (23) Δ_F, (24) Δ_N	*17*	Transf. for select	
Subqueries	5.2	(25) Δ_{Qi}	*18*	All Transf. for Q_i	
Framed Relations	5.3	(26) Δ_G, (27) Δ_{GA}	*20*	Φ_G, Φ_{GA}	*20*
Aggregate Functions	5.4	(28) Δ_A, (29) Δ_{AN}	*22*	Φ_A, Φ_{AN}	*22*
Other transformations					
Independence Predicate	3.2	IP	*4*		
Null Reduction	3.2	NR	*5*		
Nested Join Labelling	3.4	NJL	*12*		
Missing Values Outer Increment	4.1	MVOI	*14*		
Null Reduction for a Set	4.1	NRS	*15*		

Table 4. Number and coverage of the rules generated

category	type	Num Rules	SQLFpc Coverage			
			4	**10**	**100**	**1000**
Select Operators	Δ_B	261	0.38	0.38	0.77	1.15
	Δ_N	64	1.56	7.81	20.31	31.25
	Δ_T, Δ_F	513	29.82	30.80	48.93	59.84
	Sub-total:	838	18.50	19.57	31.74	39.38
Join Operators	Δ_I	80	65.00	66.25	73.75	85.00
	Δ_L, Δ_{NL}	291	22.68	28.52	48.80	59.79
	Δ_R, Δ_{NR}	397	86.65	95.21	97.23	97.48
	Sub-total:	768	60.16	66.93	76.43	81.90
Framing and Aggregate	Δ_A, Δ_{AN}	152	0.00	0.00	0.00	0.00
	Δ_G, Δ_{GA}	144	2.78	6.25	4.86	7.64
	Sub-total:	296	1.35	3.04	2.36	3.72
TOTAL:		1,902	32.65	36.12	45.22	51.00

Table 5. Number and scores of the mutants generated

Category	Num. Mutants	Mutation Score			
		4	**10**	**100**	**1000**
SQL Clause mutation (SC)	3,293	40.84	46.61	51.05	55.97
Null Mutation Operators (NL)	1,754	25.09	40.71	69.33	78.85
Operator Replacement (OR)	24,861	53.15	61.51	67.22	74.81
Identifier Replacement (IR)	162,295	65.51	73.89	81.11	88.89
TOTAL:	192,203	63.12	71.52	78.69	86.41

Table 6. Comparison of SQLFpc coverage and mutation score for generated and random test databases

Query	Query size				Developed Test Database						Random (SQLFpc Coverage)		Random (Mutation. Score)	
	CE	**NC**	**NQ**	**NT**	**TP**	**RW**	**NR**	**Cov.**	**NM**	**Mut.**	**100**	**1000**	**100**	**1000**
C_invoice_candidate_v	0	24	2	5	48	136	64	93.7	2,429	85.0	32.8	40.6	61.4	74.0
C_Invoice_LineTax_vt	15	1	5	20	63	119	89	93.5	6,194	93.1	36.0	40.4	85.4	88.4
C_RfQResponseLine_v	2	4	1	6	13	35	15	100.0	1,264	91.6	93.3	100.0	94.6	96.0
C_RfQResponseLine_vt	2	4	1	7	15	42	16	100.0	1,268	91.8	93.8	100.0	94.2	95.5
M_InOut_Candidate_v	0	17	4	6	21	44	35	85.7	1,823	90.3	5.7	37.1	4.0	65.7
RV_BPartnerOpen	1	5	2	3	8	12	14	71.4	1,101	76.5	21.4	21.4	1.7	1.7
RV_OpenItem	0	7	2	4	12	23	17	88.2	1,898	80.9	47.1	58.8	51.6	85.1
RV_WarehousePrice	2	4	1	5	13	25	16	100.0	1,422	54.5	25.0	31.3	0.5	2.5

Table 7. Summary of related work

Goal	Criterion	Scope	Integration	References
Assess adequacy	Control-flow	select	isolated	Suárez-Cabal and Tuya [31, 32]
		select	embedded	Chan and Cheung [33],
		all	embedded	Halfond and Orso [34]
	Data-flow	all	embedded	Kapfhammer and Soffa [37,40] Willmor and Embury [38] Leitao Jr. et al. [39]
	Fault-based	select	isolated	Tuya et al. [23,24] Chan et al. [35]
		select	embedded	Zhou and Frankl [22]
		select+calls	embedded	Shahriar and Zulkernine [36]
Generate test data	Specification	all	embedded	Chays et al. [41,42,44] Deng et al. [43] Willmor and Embury [45,46]
		select	isolated	Binnig et al. [47,48]
	User specified constraints	select	isolated	Zang et al. [50]
	Predicates	select	isolated	Tsai et al. [49] Khalek et al. [51]
		all	embedded	Emmi et al.. [52]
Reorder test cases	Heuristic	all	embedded	Haftmann et al. [53,54]
Select test cases	Component dependencies	all	embedded	Haraty et al. [55]
	Data-flow	all	embedded	Willmor and Embury [56]
Reduce test data	Control-flow	select	isolated	Tuya et al. [57]

FIGURES

Figure 1. Example of a parse tree for a predicate

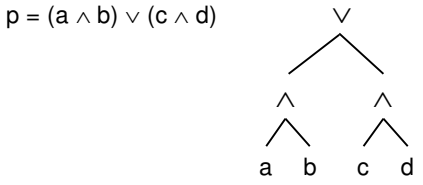


Figure 2. Graphical representation of a nested join

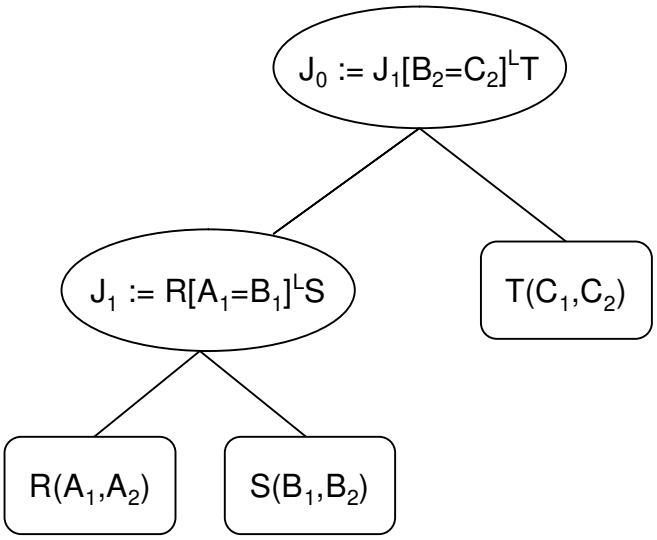


Figure 3. Algorithm to calculate the correct labelling for nested joins (NJL)

Function *NJL(L,J,i) returns label[]*
Set MVOI=∅
For each $J_k \in J$ ***do***
  Set label[k]=∅
Set label[i]=L
Add loirels(J_i) to MVOI
Loop
  Set someoneLabelled=false
  For each J_k *in J | label[k]=∅* ***do***
    If *roirels(J_k)* $\subseteq$ *MVOI* ***then***
      Set label[k]=L
      Add loirels(J_k) to MVOI
      Let someoneLabelled=true
    Else If *loirels(J_k)* $\subseteq$ *MVOI* ***then***
      Set label[k]=R
      Add roirels(J_k) to MVOI
      Let someoneLabelled=true
Until *not someoneLabelled*
For each J_k *in J | label[k]=∅* ***do***
  Set label[k]=I
Return *label*

Figure 4. Example of a nested join with four relations

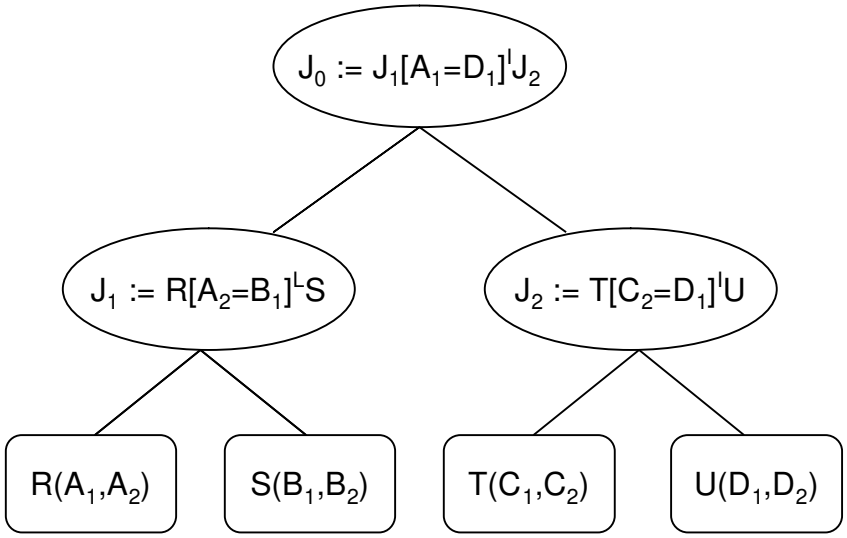


Figure 5. Architecture of the SQLFpc set of tools

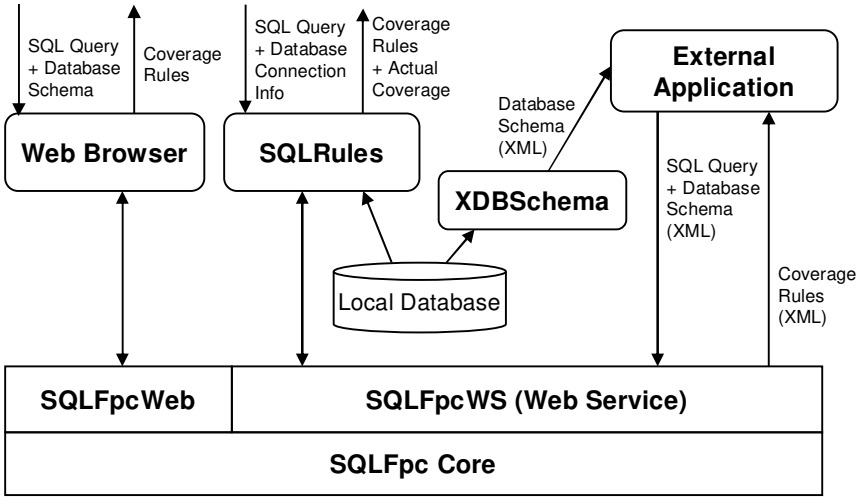
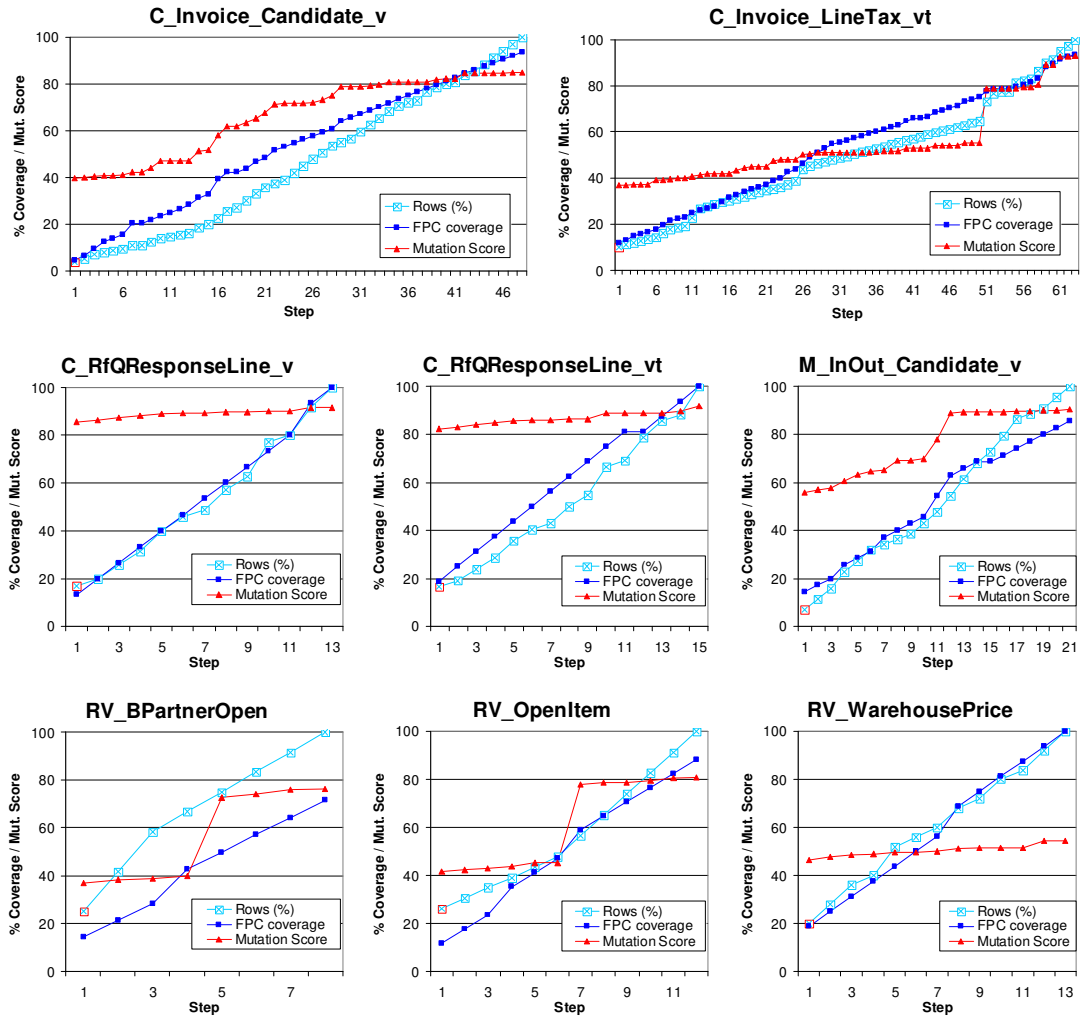


Figure 6. Evolution of the rows added to the database, coverage and mutation score



APPENDIX A

This appendix includes detailed information about the set of queries used in the Case Study, their full predicate coverage, mutation scores and performance:

- Characterization of the queries (Metrics) - NT: number of tables used, NQ; number of queries (SELECT clauses), NC: Number of conditions in WHERE and HAVING clauses, CE: number of case expressions.
- Coverage (SQLFpc Coverage/Mutation Score) - NR: Number of rules/mutants generated, 4, 10, 100, 1000: Coverage/mutation score for each database size.
- Performance (SQLFpc/Mutation execution time) - GT: User time spent on the generation of rules/mutants, 4, 10, 100, 1000: execution times for all rules against the database with the size indicated.

	Metrics				SQLFpc Coverage (%)					Mutation Score (%)					SQLFpc Execution Time (sec.)					Mutation Execution Time (sec.)				
Query Name	CE	NC	NQ	NT	NR	4	10	100	1000	NR	4	10	100	1000	GT	4	10	100	1000	GT	4	10	100	1000
AD_ChangeLog_v	0	0	1	4	5	80.0	80.0	100	100	268	4.5	4.1	88.8	92.2	0.2	0.2	0.2	0.3	0.3	0.5	3.4	3.1	4.9	51.2
AD_Field_v	0	2	1	7	12	41.7	83.3	100	100	1,961	0.1	92.2	97.4	97.8	0.7	1.9	1.8	1.0	1.4	4.9	100.5	102.5	115.3	176.2
AD_Field_vt	0	2	1	8	14	35.7	57.1	64.3	85.7	2,143	0.0	88.6	94.4	97.9	0.5	0.7	0.7	0.6	1.0	5.9	96.2	65.4	72.9	135.5
AD_Org_v	0	0	1	3	4	75.0	50.0	75.0	75.0	359	93.9	93.9	95.3	95.3	0.2	0.4	0.5	0.4	0.5	0.6	4.9	3.9	5.1	7.9
AD_Tab_v	0	2	1	2	4	50.0	100	100	100	1,114	5.4	96.9	98.2	98.4	0.2	0.2	0.2	0.2	0.3	1.8	21.1	11.1	11.6	15.7
AD_Tab_vt	0	2	1	3	5	40.0	100	100	100	1,268	4.7	89.5	98.3	98.3	0.2	0.2	0.1	0.2	0.2	2.2	17.5	14.9	17.8	24.8
AD_User_Roles_v	0	0	1	3	3	100	100	100	100	109	92.7	93.6	94.5	95.4	0.2	0.4	0.4	0.4	0.5	0.3	0.6	0.8	0.8	1.5
AD_Window_vt	0	1	1	2	3	100	100	100	100	213	88.3	93.4	95.3	95.3	0.2	0.2	0.3	0.4	0.2	0.3	2.6	1.2	1.4	2.1
C_Dunning_Header_v	1	0	1	11	25	36.0	36.0	36.0	44.0	1,599	0.4	0.4	0.4	81.6	0.6	1.8	1.9	2.0	3.1	5.6	71.6	83.2	86.0	185.4
C_Dunning_Header_vt	1	0	1	12	28	32.1	32.1	32.1	35.7	1,728	0.3	0.3	0.3	73.7	0.7	1.8	1.5	1.7	2.5	6.4	78.8	77.7	89.5	185.4
C_Dunning_Line_v	16	0	1	6	40	75.0	65.0	100	100	3,449	97.0	75.2	98.4	98.4	0.5	2.0	1.4	2.1	2.8	13.0	141.7	142.6	175.9	254.0
C_Dunning_Line_vt	19	1	1	6	48	60.4	14.6	29.2	79.2	3,587	71.7	75.1	95.4	98.4	0.8	2.3	1.6	1.9	2.7	14.7	113.0	116.8	177.7	220.9
C_Invoice_Candidate_v	0	24	2	5	64	1.6	14.1	32.8	40.6	2,429	1.6	40.6	61.4	74.0	5.5	8.0	6.4	6.2	11.6	9.5	171.8	170.3	91.4	216.0
C_Invoice_Header_v	1	0	1	11	23	56.5	60.9	91.3	91.3	2,868	94.7	96.1	96.6	96.6	0.7	2.4	2.4	2.8	3.6	11.0	254.8	214.1	258.2	371.2
C_Invoice_Header_vt	1	0	1	11	27	33.3	37.0	40.7	40.7	2,982	86.0	76.4	2.1	87.9	0.7	1.7	2.1	2.1	2.4	11.6	171.2	159.2	175.0	244.3
C_Invoice_LineTax_v	15	1	5	20	89	28.1	24.7	36.0	40.4	6,194	54.4	73.7	85.4	88.4	1.2	8.9	5.2	5.0	6.3	50.0	965.9	823.5	942.0	1,819.7
C_Invoice_LineTax_vt	18	5	5	22	108	23.1	25.0	28.7	30.6	6,809	54.3	46.3	57.2	63.6	1.5	8.8	6.0	6.3	7.4	54.2	1,173.9	1,108.8	1,968.6	7,895.1
C_Invoice_v	8	3	2	5	16	37.5	37.5	75.0	75.0	4,449	34.9	35.6	73.5	74.5	0.4	0.5	0.4	0.4	0.5	19.0	226.9	230.8	263.6	568.3
C_Invoice_v1	5	0	1	2	6	66.7	66.7	66.7	66.7	2,235	67.1	67.9	68.2	68.4	0.2	0.2	0.2	0.2	0.2	5.3	54.9	56.8	68.6	188.3
C_InvoiceLine_v	2	1	1	1	7	28.6	28.6	42.9	42.9	1,629	80.2	80.7	81.6	81.9	0.2	0.5	0.4	0.5	0.6	3.0	71.2	71.9	85.6	179.8
C_Order_Header_v	1	0	1	15	34	58.8	61.8	91.2	91.2	4,758	94.5	95.4	96.0	96.1	1.2	7.1	6.5	7.5	9.2	24.8	773.5	772.3	777.8	1,075.6
C_Order_Header_vt	1	0	1	15	38	39.5	36.8	39.5	39.5	4,890	1.3	1.3	88.2	89.7	1.1	7.5	4.1	4.6	5.9	25.3	465.3	477.5	506.0	779.5
C_Order_LineTax_v	15	0	4	18	83	21.7	22.9	33.7	31.3	5,636	48.5	55.0	84.2	85.1	1.2	7.5	5.2	5.6	6.9	38.1	702.7	718.9	830.8	1,575.5
C_Order_LineTax_vt	18	2	4	20	101	19.8	21.8	24.8	26.7	6,145	45.8	51.6	60.4	62.6	1.4	8.3	7.5	7.5	8.6	45.5	1,084.2	952.4	1,668.2	6,375.9
C_Payment_v	6	0	1	1	2	100	100	100	100	3,853	97.6	98.7	98.9	98.9	0.2	0.1	0.2	0.2	0.2	9.9	37.9	41.3	43.3	46.4
C_PaySelection_Check_v	0	0	1	5	7	71.4	71.4	71.4	100	838	1.7	1.4	2.0	97.5	0.3	0.4	0.4	0.4	0.5	1.9	14.2	14.4	13.8	27.6
C_PaySelection_Check_vt	0	0	1	6	8	62.5	62.5	62.5	100	959	1.8	1.6	2.1	97.5	0.3	0.5	0.3	0.3	0.4	2.3	21.9	22.0	21.5	91.2
C_PaySelection_Remittance_v	0	0	1	2	2	100	100	100	100	433	95.8	96.1	96.3	96.3	0.2	0.2	0.1	0.1	0.1	1.2	3.2	3.2	4.2	5.7
C_PaySelection_Remittance_vt	0	0	1	3	3	100	100	100	100	472	95.8	96.0	96.2	96.2	0.2	0.1	0.1	0.1	0.1	0.8	4.5	4.0	9.7	42.0
C_Project_Details_v	2	1	1	3	7	42.9	42.9	71.4	100	879	0.7	0.8	91.9	97.4	0.2	0.2	0.2	0.3	0.3	1.6	9.6	9.1	12.3	13.4
C_Project_Details_vt	2	1	1	4	8	50.0	50.0	75.0	100	927	0.6	0.8	91.9	97.1	0.3	0.2	0.2	0.2	0.2	2.1	11.1	11.8	16.5	24.3
C_Project_Header_v	1	0	1	12	29	55.2	58.6	89.7	93.1	2,950	94.1	96.2	96.6	96.7	0.8	3.3	3.1	3.3	4.6	11.3	264.5	265.5	313.9	444.0
C_Project_Header_vt	1	0	1	12	30	43.3	60.0	83.3	93.3	2,929	94.1	96.2	96.6	96.7	0.7	2.3	2.1	2.5	3.7	11.1	180.6	181.5	235.4	374.6
C_RfQResponse_v	0	0	1	7	13	53.8	53.8	53.8	92.3	1,044	1.9	1.8	2.2	93.5	0.3	0.5	0.6	0.6	0.9	2.9	26.9	26.7	27.4	61.1

C_RfQResponse_vt	0	0	1	8	14	57.1	57.1	57.1	92.9	1,111	2.0	1.9	2.3	93.7	0.5	0.6	0.5	0.6	1.3	3.2	27.7	28.2	33.8	221.8
C_RfQResponseLine_v	2	4	1	6	15	20.0	46.7	93.3	100	1,264	0.2	72.1	94.6	96.0	0.9	0.5	0.5	0.6	0.9	3.0	30.0	31.9	37.0	70.6
C_RfQResponseLine_vt	2	4	1	7	16	25.0	50.0	93.8	100	1,268	0.2	71.8	94.2	95.5	0.6	0.6	0.6	0.7	1.0	3.2	34.9	38.4	60.0	222.4
C_RfQResponseLineQty_v	0	2	1	3	5	80.0	100	100	100	349	92.6	93.4	94.0	94.0	0.2	0.1	0.2	0.1	0.2	0.6	2.4	2.8	2.9	8.6
C_RfQResponseLineQty_vt	0	2	1	4	6	83.3	100	100	100	388	92.0	92.8	93.3	93.3	0.2	0.1	0.3	0.2	0.2	0.6	4.0	3.5	5.6	23.0
GL_JournalLine_Acct_v	0	1	1	1	2	100	100	100	100	1,219	96.1	96.6	97.4	97.4	0.2	0.2	0.2	0.2	0.2	1.4	10.1	10.2	10.9	18.7
M_InOut_Candidate_v	0	17	4	6	35	0.0	0.0	5.7	37.1	1,823	0.1	0.3	4.0	65.7	0.5	1.1	0.9	0.9	1.1	5.5	51.0	48.6	49.6	58.0
M_InOut_Header_v	0	0	1	9	17	64.7	70.6	88.2	88.2	2,089	95.0	95.5	96.0	96.1	0.6	1.5	1.0	1.3	1.7	7.0	98.0	5.3	121.3	189.3
M_InOut_Header_vt	0	0	1	9	20	55.0	50.0	55.0	65.0	2,183	95.1	95.6	96.2	96.2	0.5	0.9	0.8	1.1	1.7	7.8	82.3	78.5	111.0	146.5
M_InOut_Line_v	6	0	2	14	41	29.3	36.6	43.9	68.3	3,698	41.9	85.7	89.0	91.7	0.7	1.8	1.5	1.7	3.0	16.5	281.3	296.8	392.4	1,261.0
M_InOut_Line_vt	9	0	2	16	54	24.1	29.6	33.3	44.4	4,072	41.5	45.3	48.5	89.3	0.8	2.8	2.6	2.2	2.9	19.8	375.4	393.2	553.0	1,549.6
M_InOut_LineConfirm_v	2	0	1	7	13	53.8	84.6	100	100	1,696	93.8	97.5	98.0	98.0	0.3	0.5	0.5	0.6	0.8	4.2	46.8	48.1	58.2	121.2
M_InOut_LineConfirm_vt	3	0	1	8	19	47.4	52.6	57.9	68.4	1,917	92.1	97.3	96.7	97.8	0.4	0.7	0.7	0.8	1.0	5.2	59.7	60.9	80.8	129.4
M_InOutConfirm_v	0	0	1	3	3	100	100	100	100	1,015	97.3	97.9	98.2	98.2	0.2	0.1	0.2	0.1	0.2	2.1	12.6	13.0	16.6	31.2
M_InOutConfirm_vt	0	0	1	3	4	75.0	75.0	100	100	999	97.2	97.9	98.2	98.2	0.2	0.1	0.2	0.1	0.1	2.3	11.5	12.5	18.2	26.1
M_InOutLineMA_v	0	0	2	3	2	100	100	100	100	580	95.2	95.5	95.7	95.7	0.2	0.1	0.2	0.1	0.2	0.8	4.8	5.2	4.9	11.5
M_InOutLineMA_vt	0	0	2	3	2	100	100	100	100	580	95.2	95.5	95.7	95.7	0.2	0.1	0.1	0.1	0.1	0.7	1.6	1.7	2.9	9.2
M_MovementLineMA_v	0	0	2	3	2	100	100	100	100	552	94.4	94.7	95.1	95.1	0.2	0.1	0.1	0.1	0.2	0.8	3.9	4.1	5.4	11.3
M_MovementLineMA_vt	0	0	2	3	2	100	100	100	100	552	94.4	94.7	95.1	95.1	0.2	0.1	0.1	0.1	0.1	0.7	1.6	1.6	2.6	9.5
M_Transaction_v	0	0	1	7	17	47.1	58.8	64.7	64.7	1,264	69.6	79.7	89.2	96.7	0.3	0.6	0.6	0.6	1.0	2.4	29.6	30.0	39.7	92.3
R_Request_v	0	3	1	1	5	20.0	80.0	80.0	80.0	75	18.7	90.7	92.0	92.0	0.2	0.2	0.2	0.2	0.2	0.2	0.5	0.5	0.6	0.7
RV_Allocation	0	0	1	2	2	100	100	100	100	557	95.3	96.1	96.6	96.6	0.2	0.7	0.1	0.5	0.2	0.8	4.6	4.6	4.9	7.8
RV_Asset_Customer	0	2	2	2	4	75.0	100	100	100	468	92.7	96.4	97.2	97.2	0.2	0.2	0.2	0.2	0.2	0.8	3.0	3.0	3.0	9.0
RV_Asset_Delivery	0	0	1	2	2	100	100	100	100	480	95.2	97.3	97.3	97.3	0.2	0.1	0.1	0.1	0.1	0.8	3.2	3.2	5.4	6.5
RV_Asset_SumMonth	0	0	1	2	5	60.0	80.0	60.0	80.0	605	96.4	97.2	97.9	97.9	0.2	0.2	0.1	0.2	0.2	2.0	9.1	9.7	12.1	86.0
RV_BPartner	1	0	1	6	11	36.4	45.5	81.8	90.9	5,509	85.5	86.4	97.1	97.2	0.4	0.7	0.6	0.7	2.7	18.0	254.1	262.2	342.1	535.2
RV_BPartnerOpen	1	5	2	3	14	7.1	14.3	21.4	21.4	1,101	0.9	1.7	1.7	1.7	0.2	0.6	0.5	0.5	0.6	2.9	52.7	53.6	53.4	69.8
RV_C_Invoice	4	0	1	5	8	75.0	87.5	87.5	87.5	2,593	73.7	74.5	75.2	75.3	0.3	0.5	0.7	0.5	0.6	7.1	94.9	97.9	118.0	232.4
RV_C_Invoice_CustomerProdQtr	2	0	1	1	27	0.0	0.0	0.0	0.0	897	75.8	75.7	75.8	75.8	0.2	1.7	1.6	2.0	3.9	2.0	50.1	54.1	66.6	162.0
RV_C_Invoice_CustomerVendQtr	2	1	1	2	31	3.2	16.1	19.4	19.4	999	0.7	77.6	78.2	78.2	0.3	2.1	2.0	2.2	3.4	2.3	63.5	62.9	73.8	261.8
RV_C_Invoice_Day	2	0	1	1	24	0.0	0.0	0.0	0.0	806	74.4	74.4	74.6	74.6	0.2	1.5	1.4	1.5	3.1	1.9	43.3	46.0	55.5	116.9
RV_C_Invoice_Month	2	0	1	1	24	0.0	0.0	0.0	0.0	806	74.4	74.4	74.6	74.6	0.2	1.4	1.4	1.5	3.1	1.7	43.0	45.9	54.8	151.3
RV_C_Invoice_ProdMonth	2	0	1	1	30	0.0	0.0	0.0	0.0	850	75.1	74.9	75.1	75.1	0.2	1.8	1.8	2.1	4.5	1.8	47.6	50.9	63.0	151.5
RV_C_Invoice_ProductMonth	2	0	1	1	30	0.0	0.0	0.0	0.0	850	75.1	74.9	75.1	75.1	0.2	1.8	1.7	2.0	4.2	1.8	45.7	48.8	58.1	142.7
RV_C_Invoice_ProductQtr	2	0	1	1	30	0.0	0.0	0.0	0.0	850	75.1	74.9	75.1	75.1	0.2	1.8	1.8	2.0	4.2	1.8	47.2	49.7	60.2	140.4
RV_C_Invoice_ProdWeek	2	0	1	1	30	0.0	0.0	0.0	0.0	850	75.1	74.9	75.1	75.1	0.3	1.8	1.8	2.1	4.6	1.8	47.7	50.6	63.4	162.5
RV_C_Invoice_VendorMonth	2	1	1	2	31	3.2	16.1	19.4	19.4	1,000	0.7	77.5	78.1	78.1	0.3	2.2	2.0	2.3	3.5	2.3	63.9	63.3	77.6	258.9
RV_C_Invoice_Week	2	0	1	1	24	0.0	0.0	0.0	0.0	806	74.4	74.4	74.6	74.6	0.2	1.4	1.4	1.5	3.1	1.7	42.9	46.1	55.0	116.5
RV_C_InvoiceLine	5	0	1	4	16	31.3	31.3	43.8	43.8	3,477	80.3	80.6	81.1	81.1	0.3	1.0	1.1	1.1	1.3	9.8	194.6	215.4	262.2	643.6
RV_C_InvoiceTax	4	0	1	4	6	83.3	83.3	83.3	83.3	1,258	65.8	67.9	68.8	68.9	0.3	0.3	0.3	0.3	0.4	3.1	28.9	29.7	41.2	92.5
RV_C_RfQ_UnAnswered	0	2	1	2	4	50.0	100	100	100	496	83.9	95.6	96.6	96.6	0.2	0.1	0.1	0.1	0.1	0.9	4.0	4.1	4.6	5.7
RV_C_RfQResponse	0	2	1	6	8	25.0	25.0	37.5	37.5	1,451	3.9	2.2	0.5	0.1	0.3	0.3	0.3	0.3	0.4	3.4	31.2	32.4	33.3	51.9
RV_Cash_Detail	0	0	1	3	3	100	100	100	100	675	75.6	84.4	85.0	85.0	0.2	0.2	0.2	0.3	0.2	1.0	8.2	9.4	15.0	69.2
RV_Click_Month	0	0	1	2	11	27.3	36.4	36.4	45.5	114	81.6	82.5	85.1	85.1	0.2	0.5	0.2	0.2	0.4	0.3	1.3	1.3	1.9	53.5
RV_Click_Unprocessed	0	2	1	1	3	66.7	100	100	100	28	67.9	89.3	92.9	92.9	0.7	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
RV_CommissionRunDetail	6	0	1	8	22	59.1	95.5	100	100	2,043	71.7	97.0	97.7	97.7	0.5	0.9	0.9	0.9	1.6	5.8	62.9	64.1	80.7	139.5
RV_Cost	0	0	1	4	4	100	100	100	100	713	96.1	96.9	96.9	96.9	0.2	0.2	0.2	0.2	0.2	1.3	9.7	10.2	12.2	19.7
RV_CostDetail	0	0	1	4	4	100	100	100	100	843	96.4	96.4	97.3	97.3	0.2	0.6	0.2	0.2	0.3	1.5	11.2	11.2	14.1	21.2
RV_CostSummary	0	1	1	3	15	26.7	20.0	26.7	26.7	660	2.0	1.4	1.5	3.9	0.2	0.3	0.3	0.4	0.4	1.7	8.9	8.8	8.9	12.8
RV_Fact_Acct	1	0	1	5	10	60.0	70.0	70.0	70.0	2,877	91.6	92.0	92.4	92.5	0.3	0.3	0.4	0.4	0.7	6.6	82.4	83.5	103.2	153.6
RV_Fact_Acct_Day	1	0	1	1	34	0.0	0.0	0.0	0.0	2,093	87.9	88.2	88.6	88.6	0.3	0.4	0.4	0.4	1.0	5.6	28.4	24.2	39.6	79.4
RV_Fact_Acct_Period	1	0	1	1	33	0.0	0.0	0.0	0.0	2,085	88.0	88.3	88.6	88.7	0.2	0.3	0.3	0.3	0.5	5.5	20.8	23.0	27.0	49.1
RV_InOutConfirm	0	0	1	2	2	100	100	100	100	445	96.2	97.1	97.1	97.1	0.2	0.1	0.1	0.1	0.1	0.8	2.9	3.0	4.8	5.6
RV_InOutDetails	0	0	1	4	6	100	83.3	100	100	2,898	97.4	98.1	98.5	98.5	0.2	0.3	0.3	0.3	0.3	7.1	71.3	78.7	88.8	117.6
RV_InOutLineConfirm	0	0	1	4	4	100	100	100	100	935	96.1	97.0	97.2	97.2	0.2	0.1	0.1	0.2	0.2	1.6	11.7	11.7	15.8	31.7
RV_M_Requisition	0	0	1	2	2	100	100	100	100	505	96.0	95.6	96.6	96.6	0.2	0.1	0.2	0.1	0.2	0.7	4.5	3.9	3.7	6.5

RV_M_Transaction	0	1	1	3	5	80.0	100	100	100	474	94.7	95.6	96.2	96.2	0.2	0.2	0.2	0.2	0.2	0.7	4.0	3.8	6.2	10.1
RV_M_Transaction_Sum	0	1	1	1	11	18.2	18.2	18.2	18.2	201	85.1	86.1	86.1	86.1	0.2	0.1	0.2	0.2	0.3	0.4	1.2	1.6	1.6	11.1
RV_OpenItem	0	7	2	4	17	23.5	29.4	47.1	58.8	1,898	45.9	45.8	51.6	85.1	0.3	1.3	1.0	1.3	4.8	5.8	145.7	163.8	363.8	2,682.2
RV_OrderDetail	3	0	1	3	10	30.0	30.0	40.0	40.0	3,627	88.0	88.6	88.9	88.9	0.3	0.3	0.3	0.3	0.3	10.0	80.0	89.6	116.4	269.6
RV_Payment	6	0	1	1	2	100	100	100	100	3,961	95.8	96.9	97.1	97.1	0.8	0.1	0.2	0.2	0.2	11.1	58.4	80.4	83.9	112.2
RV_PrintFormatDetail	0	0	1	2	2	100	100	100	100	2,559	98.1	98.5	98.8	98.8	0.2	0.2	0.2	0.2	0.2	4.9	36.3	37.6	38.5	55.2
RV_Product_Costing	3	0	1	2	11	18.2	18.2	18.2	27.3	1,131	78.3	78.5	78.6	84.5	0.2	0.4	0.2	0.2	0.2	1.7	12.3	12.4	18.4	37.2
RV_ProjectCycle	0	0	1	6	9	88.9	88.9	100	100	2,605	1.7	78.2	78.5	78.5	0.3	0.8	0.6	0.7	0.8	7.5	109.2	133.8	213.4	1,040.0
RV_ProjectLineIssue	9	0	1	3	24	45.8	50.0	45.8	50.0	2,093	64.9	65.6	60.6	66.0	0.3	0.8	0.4	0.5	0.6	4.0	38.5	44.0	54.9	162.8
RV_RequestUpdates	0	1	7	12	20	65.0	75.0	85.0	100	1,626	60.1	61.5	62.7	96.0	0.3	0.3	0.3	0.3	0.4	4.9	56.3	78.9	76.8	109.2
RV_RequestUpdates_Only	0	1	1	1	10	40.0	50.0	40.0	50.0	88	77.3	88.6	77.3	88.6	0.2	0.3	0.3	0.4	0.6	0.2	1.9	2.3	3.0	4.4
RV_Storage	0	0	1	4	4	100	100	100	100	1,247	95.9	96.2	97.3	97.4	0.2	0.2	0.2	0.3	0.3	2.8	30.0	30.8	35.8	48.6
RV_Transaction	1	0	1	10	20	60.0	75.0	95.0	100	2,337	90.2	93.1	94.1	95.9	0.5	1.0	0.9	1.0	1.7	6.3	81.4	82.3	94.5	191.7
RV_UnPosted	0	15	15	16	31	54.8	61.3	100	100	4,381	93.3	94.4	97.5	97.5	0.4	1.0	0.4	0.4	0.5	21.1	192.6	197.0	227.9	508.8
RV_WarehousePrice	2	4	1	5	16	18.8	25.0	25.0	31.3	1,422	0.0	0.5	0.5	2.5	0.4	0.8	0.8	1.0	0.9	4.1	58.4	55.8	56.0	62.2
T_InvoiceGL_v	0	0	1	3	3	100	100	100	100	3,065	97.7	97.8	98.3	98.3	0.2	0.1	0.2	0.2	0.2	6.3	52.7	55.0	66.2	90.3
T_InvoiceGL_vt	0	0	1	3	3	100	100	100	100	3,065	97.7	97.8	98.3	98.3	0.2	0.1	0.1	0.1	0.1	6.1	53.6	54.3	66.0	92.1
Total general	226	127	155	564	1,902	32.6	36.1	45.2	51.0	192,203	63.1	71.5	78.7	86.4	45.9	128.5	107.6	115.3	170.5	692.4	11,024.7	10,764.8	14,147.7	37,543.6

APPENDIX B

This appendix includes the SQL of a query used in the Case Study for a detailed test: C_Invoice_Candidate_v

```
SELECT   o.AD_Client_ID, o.AD_Org_ID, o.C_BPartner_ID, o.C_Order_ID,
     o.DocumentNo, o.DateOrdered, o.C_DocType_ID,
     SUM((l.QtyOrdered-l.QtyInvoiced)*l.PriceActual) AS TotalLines
FROM C_Order o
  INNER JOIN C_OrderLine l ON (o.C_Order_ID=l.C_Order_ID)
  INNER JOIN C_BPartner bp ON (o.C_BPartner_ID=bp.C_BPartner_ID)
  LEFT OUTER JOIN C_InvoiceSchedule si ON (bp.C_InvoiceSchedule_ID=si.C_InvoiceSchedule_ID)
WHERE    o.DocStatus IN ('CO','CL','IP')     -- Standard Orders are IP
     --  not Offers and open Walkin-Receipts
     AND o.C_DocType_ID IN (SELECT C_DocType_ID FROM C_DocType
          WHERE DocBaseType='SOO' AND DocSubTypeSO NOT IN ('ON','OB','WR'))
     --  we need to invoice
     AND l.QtyOrdered <> l.QtyInvoiced
     --
     AND (--  Immediate
          o.InvoiceRule='I'
          --  Order complete ** not supported **
          OR  o.InvoiceRule='O'
          --  Delivery
          OR  (o.InvoiceRule='D' AND l.QtyInvoiced<>l.QtyDelivered)
          --  Order Schedule, but none defined on Business Partner level
          OR  (o.InvoiceRule='S' AND bp.C_InvoiceSchedule_ID IS NULL)
          --  Schedule defined at BP
          OR  (o.InvoiceRule='S' AND bp.C_InvoiceSchedule_ID IS NOT NULL AND
                ( --  Daily or none
                     (si.InvoiceFrequency IS NULL OR si.InvoiceFrequency='D')
                  --  Weekly
                  OR  (si.InvoiceFrequency='W')
                  --  Bi-Monthly
                  OR  (si.InvoiceFrequency='T'
                       AND (TRUNC(o.DateOrdered) <= firstOf(getdate(),'MM')+si.InvoiceDayCutoff-1
                            AND TRUNC(getdate()) >= firstOf(o.DateOrdered,'MM')+si.InvoiceDay-1)
                       OR  (TRUNC(o.DateOrdered) <= firstOf(getdate(),'MM')+si.InvoiceDayCutoff+14
                            AND TRUNC(getdate()) >= firstOf(o.DateOrdered,'MM')+si.InvoiceDay+14)
                       )
                  --  Monthly
                  OR  (si.InvoiceFrequency='M'
                       AND TRUNC(o.DateOrdered) <= firstOf(getdate(),'MM')+si.InvoiceDayCutoff-1  -- after cutoff
                       AND TRUNC(getdate()) >= firstOf(o.DateOrdered,'MM')+si.InvoiceDay-1)       -- after invoice day
          )    )    )
GROUP BY o.AD_Client_ID, o.AD_Org_ID, o.C_BPartner_ID, o.C_Order_ID, o.DocumentNo, o.DateOrdered, o.C_DocType_ID
```