Multi-RQP – Generating Test Databases for the Functional Testing of OLTP Applications

Carsten Binnig
SAP AG
carsten.binnig@sap.com

Donald Kossmann
ETH Zurich
kossmann@inf.ethz.ch

Eric Lo
The Hong Kong Polytechnic University
ericlo@comp.polyu.edu.hk

ABSTRACT

OLTP applications usually implement use cases which execute a sequence of actions whereas each action usually reads or updates only a small set of tuples in the database. In order to automatically test the correctness of the different execution paths of the use cases implemented by an OLTP application, a set of test cases and test databases needs to be created.

In this paper, we suggest that a tester specifies a test database individually for each test case using SQL as a declarative test database specification language. Moreover, we also discuss the design of a database generator which creates a test database based on such a specification. Consequently, our approach allows to generate a tailor-made test database for each test case and to bundle them together for the test case execution phase.

1. INTRODUCTION

OLTP applications usually implement use cases which execute a sequence of actions whereas each action usually reads or updates only a small set of tuples in the database. As an example, think of an online library. One potential use case of such an application is that a user wants to borrow a book. The sequence of actions which is implemented by that use case could be as follows:

1. The user enters the ISBN of the book (where the ISBN is unique).
2. The system shows the details of that book.
   - Exception 1: The book is borrowed by another user. The system denies the request.
   - Exception 2: The book belongs to the closed stack of the library. The system denies the request.
3. The user enters personal data (username, password) and confirms that she wants to borrow the book.
4. The system checks the user data and updates the database.
   - Exception 3: The user has entered a wrong username or password. The system denies the request.
   - Exception 4: There are charges on the user account that exceed a certain limit. The system denies the request.

Functional testing the implementation of such a use case means that we have to check the conformance of the implementation with the specification of the functionality [3] (i.e., the use case). Consequently, we need to create a set of test cases to test the correctness of the different execution paths of a use case. In the following we show a possible set of test cases:

DBTest '08, June 13, 2008, Vancouver, BC, Canada

- *Test Case 1:* The user wants to borrow a book with a particular ISBN that is already borrowed by another user.
- *Test Case 2:* The user wants to borrow a book with a particular ISBN but the book belongs to the closed stack.
- *Test Case 3:* The user wants to borrow a book with a particular ISBN and enters a wrong username or password.
- *Test Case 4:* The user wants to borrow a book with a particular ISBN but there are charges on her account that exceed a certain limit.
- *Test Case 5:* The user borrows a book with a particular ISBN successfully.

In order to execute all these test cases, one or more test databases need to be created which comprise different types of books and different user accounts. For example, in order to execute *Test Case 2* the database should include a book which belongs to the closed stack.

Currently, there are a number of commercial and academic tools available (e.g., [2, 1, 7, 13, 10, 12, 8]) which generate test databases for a given database schema. Beside the database schema, some tools also support the input of the table sizes, data repositories and additional constraints used for data instantiation (e.g., statistical distributions of individual attributes, value ranges). Unfortunately, all these tools suffer from the problem that the generated test databases often do not comprise the data characteristics sufficient to execute a given set of test cases. The reason is that these tools take constraints on the complete database state as input (e.g., table sizes and value distributions of individual attributes) which are not suitable to express the needs of the individual test cases.

In this paper, we suggest that a tester specifies the test database for each test case individually using declarative test database specification language. For example, in order to specify a test database for *Test Case 4* above, the test database needs to comprise a book with a particular ISBN which does not belong to the closed stack (i.e., the attribute $b\_closedstack$ must have the value '$false$') and a user whose charges exceed a certain limit (e.g. $20). The desirable database state, can be specified by multiple queries and the corresponding expected query results (e.g., the queries and expected results shown in the following example). A database generator then creates a test database for a given database schema [1] which returns the given expected result for each query of the specification. Consequently, this approach allows to generate a tailor-made test database for each test case and to bundle them together for the test case execution phase. Moreover, the tester can focus on the data that is relevant for *Test Case 4* (e.g., the values for b_isbn and $b\_closedstack$ specified by Q_1 and R_1) and she does not have to take care of the irrelevant data (e.g., the values for b_price and b_title).

[1]The database schema for all examples in this paper is shown in Figure 3 (a) on Page 6.

```
Q1 :  SELECT b_closedstack FROM book
      WHERE b_isbn='0130402648'
R1 :  {<'false'>}
Q2 :  SELECT u_pasword, u_charges FROM user
      WHERE u_name='test'
R2 :  {<test, 20.0>}
```

The contributions of this paper can be summarized as follows: (1) In order to generate a test database for such a specification, we study the problem of Multi-RQP (or MRQP for short). MRQP is an extension to RQP [4]: While RQP takes as input one SQL SELECT query and the corresponding expected query results, and generates a test database that returns the expected results for that given query, MRQP supports a more general case and takes as input a *set* of SQL SELECT queries and the *corresponding expected query results* and generates a test database that returns the expected results for all the given queries. However, MRQP is undecidable for arbitrary SQL SELECT queries. (2) Thus, in order to generate test databases for a test case of an OTLP application, we propose a new database specification language called MSQL. MSQL is a pure subset of SQL. Using MSQL the tester can easily formulate queries for the test database specification that satisfy certain restrictions so that MRQP becomes decidable and can be solved efficiently while the tester can still specify any test database for a given schema. (3) Using the specified queries and expected results, we discuss how a test database can be automatically generated by MRQP which is adequate to support the execution of a particular test case.

The remainder of this paper is organized as follows: Section 2 discusses the problem statement of MRQP and defines the general restrictions on the input queries of MRQP . Section 3 then introduces the test database specification language MSQL. Moreover, we also show a complete example of MRQP using MSQL. Finally, Section 4 presents the related work and Section 5 contains conclusions and suggestions for future work.

2. MRQP OVERVIEW

In this section we first study the decidability of MRQP. Therefore, we present the general problem statement of MRQP and show that MRQP is undecidable for arbitrary SQL queries. Afterwards, we introduce some restrictions on the input queries of MRQP such that MRQP becomes decidable and can be solved efficiently. Finally, we illustrate a procedure which solves MRQP under these restrictions.

2.1 Problem Statement and Decidability

As mentioned before, this paper addresses the following problem: Given a set of arbitrary SQL SELECT queries $Q = \{Q_1, ..., Q_n\}$, a set of expected results $R = \{R_1, ..., R_n\}$ of these queries, and the database schema S of a relational database (including integrity constraints), find a database instance D so that $R_i = Q_i(D)$ is valid for all $1 \leq i \leq n$ and D is compliant with S and its integrity constraints. There may exist many different database instances D that satisfy these criteria. In this paper, it is the goal to find one viable database instance.

The decision problem (based on the problem statement above) which asks whether a database instance D exists or not that satisfies the schema S and returns $R_i = Q_i(D)$ for all $1 \leq i \leq n$ is thus called the *MRQP decision problem*. Obviously, the MRQP decision problem cannot be decidable because RQP is not decidable for arbitrary SQL queries.

2.2 MRQP Restrictions

As we have shown before, MRQP is undecidable for arbitrary SQL queries. Consequently, we restrict the input queries in Q so that the MRQP decision problem becomes decidable. The basic restriction is that MRQP supports only those query classes as input such that RQP becomes decidable for each individual query [4].

Moreover, we introduce a further restriction on the query set Q which requires that Q must be *RQP-disjoint*. Under that restriction MRQP can be solved efficiently by first generating one individual test database for each query $Q_i \in Q$ using RQP and then taking the union over all these individual test databases. However, in some cases it is cumbersome to specify the intended test database by only using an RQP-disjoint query set Q. Therefore, we introduce a relaxation of that restriction which enables a tester to specify the test database in a more elegant way by creating *query refinements* for the individual queries $Q_i \in Q$.

2.2.1 RQP-disjoint Queries

In order to solve MRQP efficiently, we require that the input query set Q is *RQP-disjoint*.

DEFINITION 2.1. *(RQP-disjoint Queries:) A set of queries Q is RQP-disjoint iff all possible pairs (Q_j, Q_k) with $j \neq k$ are RQP-disjoint. Two queries Q_j and Q_k in Q with $j \neq k$ are RQP-disjoint, iff the view specified by query Q_j is update independent from any update (i.e., INSERT statement) that could be generated by RQP for the query Q_k and any possible expected result R_k of that query and vice versa.*

As an example, look at the following two SQL queries Q_1 and Q_2 and the corresponding expected results R_1 and R_2 which specify the test database for *Test Case 3* in Section 1. This test case requires a test database which comprises a book with a particular ISBN that does not belong to the closed stack and a user with a distinct user name and a password which is different from a given password (that is used as input value for the test case). The two queries Q_1 and Q_2 are RQP-disjoint because Q_2 is update independent from any INSERT statement that could be generated by an RQP processor for Q_1 any expected result of that query (e.g., Q_1 is update independent from the INSERT statement I_1 which is generated for Q_1 and R_1 by an RQP processor) and vice versa (e.g., the view defined by Q_1 is update independent from I_2).

```
Q1 :  SELECT b_closedstack FROM book
      WHERE b_isbn='0201485419'
R1 :  {<'false'>}
I1 :  INSERT INTO book
      (b_id, b_title, b_price, b_isbn, b_closedstack)
      VALUES (1, 'TitleB', 100.0, '0201485419', 'false')
Q2 :  SELECT COUNT(*) FROM user
      WHERE u_name='test' AND u_password!='test'
R2 :  {<1>}
I2 :  INSERT INTO user
      (u_id, u_name, u_password, u_charges)
      VALUES (1, 'test', 'test1', 0.0)
```

If the queries in Q are RQP-disjoint, we can generate a test database by calling the RQP processor separately for each query and the corresponding expected result (i.e., $RQP(Q_1, R_1, S) = D_1$, ..., $RQP(Q_n, R_n, S) = D_n$)[2]. Afterwards, we take the union of all the individual test databases to create the final test database (i.e., $D = D_1 \cup ... \cup D_n$)[3]. Continuing the example above: In order to generate a test database for the two RQP-disjoint queries Q_1 and Q_2 and the two expected results R_1 and R_2, we first generate two

[2]In this paper we call the RQP processor as an external function which takes a query Q, an expected result R, and a database schema S as input and generates a database D which satisfies S and returns $Q(D) = R$.

[3]The $\cup$ operator here creates the union over all tables of the database schema S.

individual databases D_1 and D_2. Consequently, the test database D_1 comprises one book with the given ISBN and the value specified for the attribute $b\_closedstack$ (i.e., D_1 is created by I_1) and test database D_2 comprises the user account with the given username and a password which is not equal to the input value '$test$' (i.e., D_2 is created by I_2). Subsequently, the final test database D is $D = D_1 \cup D_2$.

In order to make sure that the final database D which is generated for an RQP-disjoint query set fulfills the *primary-key* and *unique* constraints in the database schema S, MRQP has to make sure that the individual RQP calls assign unique values to the attributes in S that are bound by such a constraint for all queries in Q and the corresponding expected results in R.

Using an RQP-disjoint query set as input of MRQP, the user can specify any database instance for a given schema S. In order to show that this is possible, we assume that a tester creates one query per table which reads all tuples (e.g., `SELECT * FROM orders`) and the expected results of these queries. Using these queries and the expected results the tester can obviously control all attribute values individually for each tuple in every table of the database schema S and thus specify any database instance.

2.2.2 Query Refinements

Using only an RQP-disjoint query set to specify the intended test database for a test case can sometimes be cumbersome for the tester. For example, assume the tester wants to specify a test database which should comprise five books with a total sum of prices which is $1000 while one of these books should have the price $100 and the title '$TitleA$'.

Unfortunately, there is no elegant way to specify such a test database by using only an RQP-disjoint query set: (1) The first possibility is that the tester specifies one query (i.e, `SELECT b_price, b_title FROM book`) and defines an expected result which holds the values for the attributes b_price and b_title of all books (while the tester has to manually take care that the total sum is $1000 and she also has to define the titles for four out of five books that are not relevant for the test case). (2) Another possibility is that the tester specifies two individual SQL queries while one query specifies the one book which has the price of $100 and the title '$TitleA$' (as shown by the query Q_1 and the expected result R_1 in the following example) and the other query specifies the remaining four books (as shown by query Q_2 and the expected result R_2 in the following example). However, in that case the tester has to manually adjust the query Q_2 and the expected result R_2 so that the total sum for the four remaining books is $900 and none of these books uses the same ISBN as the book with the price of $100 (i.e., the selection predicate of Q_2 must be $b\_isbn! =$ '0130402648').

```
Q1 :  SELECT b_price, b_title FROM book
      WHERE b_isbn='0130402648'
R1 :  {<100.0, 'TitleA'>}
Q2 :  SELECT SUM(b_price), COUNT(*) FROM book
      WHERE b_isbn!='0130402648'
R2 :  {<900.0, 4>}
```

A more elegant solution is that in addition to the RQP-disjoint set of queries Q and the expected results R we allow the user to define a *query refinement* for each query $Q_i \in Q$. The intuition is that a query refinement F_i for a query Q_i defines more attribute values for a subset of tuples that are specified by Q_i and R_i. That is, a query refinement F_i refines a query Q_i. In the following we give a more formal definition and show how MRQP can generate the test database if some queries $Q_i \in Q$ are refined by a query refinement.

DEFINITION 2.2. *(Query Refinement:) A query refinement F_i for one query $Q_i \in Q$ is a set of RQP-disjoint queries $F_i = \{F_{i1}, \ldots, F_{in}\}$ plus the expected results RF_i for each query in F_i; i.e., $RF_i = \{RF_{i1}, \ldots, RF_{in}\}$ where Q_i is update dependent (=opposite of update independent) of all `INSERT` statements that could be generated by RQP for any query $F_{ij} \in F_i$ and an arbitrary expected result $RF_{ij} \in RF_i$ of that query. Moreover, $baseAttr(R_i) \subseteq baseAttr(RF_{ij})$ must hold for all $RF_{ij} \in RF_i$ (R_i is the expected result of Q_i and $baseAttr(R_i)$ is a function that extracts the names of the attributes in the database schema S that participate in the expected result R_i). Furthermore, for each query $F_{ij} \in F_i$ the user can recursively specify further query refinements.*

A simple query refinement for the example above is shown by the following query Q_1 and the refinement given by $F_1 = \{F_{11}\}$. While Q_1 specifies the total sum of prices for all books, F_1 specifies the price and the title of one book with a particular ISBN number. Obviously, Q_1 is update dependent from any `INSERT` statement that could be generated by RQP for each query in F_1 and some arbitrary expected results (e.g., Q_1 is update dependent from the `INSERT` statement IF_{11} which is generated by RQP for the expected result RF_{11} and the query F_{11}). Moreover, $baseAttr(R_1) = \{b\_price\}$ is a subset of $baseAttr(RF_{11}) = \{b\_price, b\_title\}$. Thus, $F_1 = \{F_{11}\}$ is a query refinement for query Q_1.

```
Q1 :    SELECT SUM(b_price), COUNT(*) FROM book
R1 :    {<1000.0, 5>}
F11 :   SELECT b_price, b_title FROM book
        WHERE b_isbn='0130402648'
RF11 :  {<100, 'TitleA'>}
IF11 :  INSERT INTO book
        (b_id, b_title, b_price, b_isbn, b_closedstack)
        VALUES (1, 'TitleA', 100.0, '0130402648', 'false')
```

In the following we illustrate how MRQP can generate a test database for a query Q_i of an RQP-disjoint query set Q which is refined by a query refinement F_i and its expected results RF_i. A general solution how to generate a test database for a RQP-disjoint query set Q where some queries $Q_i \in Q$ can be recursively refined by a query refinement is shown in the next Section 2.3. The idea presented here is similar to the one shown for an RQP-disjoint query set. MRQP first generates one test database for Q_i and another one for F_i by calling an RQP processor individually for Q_i and F_i and taking the union of both test databases. However, before the test database for the query Q_i and its expected result R_i can be generated by an RQP processor, MRQP has to adjust Q_i and R_i w.r.t. the query refinement F_i and its expected results RF_i. The details of this process are described in the sequel.

Firstly, MRQP generates a test database DF_i for the query refinement F_i of query Q_i and the expected results RF_i of the refinement F_i as described in Section 2.2.1 for any RQP-disjoint set of queries. Afterwards, MRQP *adjusts* the expected result R_i of the query Q_i which is refined by F_i with respect to the generated test database DF_i by executing $R_i' = R_i \ominus Q_i(DF_i)$. The operator $\ominus$ is called the *Adjust* operator and its implementation depends on the type of query Q_i. In general, the $\ominus$ operator "removes" those tuples from the expected result R_i that are already specified by the queries in the refinement F_i and the expected results RF_i and thus do not have to be generated for the query Q_i and the expected result R_i anymore. Moreover, in addition to the expected result R_i, we also have to adjust the query Q_i (which results in Q_i') so that RQP generates no tuples for Q_i' and the adjusted expected result R_i' that would be returned by any query in the refinement F_i. A detailed description of the implementation of the *Adjust* operator and the function which adjusts the query Q_i will be given in Section 3

MRQP(Queries Q, Results R, Schema S, Refinements (F_i, RF_i))
Output: database D

```
(1)  D=∅ //Generate an empty DB
(2)  FOR EACH Query Q_i in Q
(3)   R_i= R.get(i) //Extract expected result
(4)   DF_i=∅
(5)   IF(Q_i has a Query Refinement (F_i, RF_i))
(6)    DF_i=MRQP(F_i, RF_i, S) //Generate DB for F_i
(7)    R_i=R_i ⊖ Q_i(DF_i) //Adjust result
(8)    Q_i=AdjustQuery(Q_i, F_i) //Adjust query
(9)   END IF
(10)  IF(D=D∪RQP(Q_i,R_i,S)∪DF_i returns ERROR)
(11)   RETURN ERROR
(12)  END IF
(13) END FOR
(14) RETURN D
```

Figure 1: Function *MRQP*

for all query classes supported in in the database specification language MSQL. Subsequently, MRQP generates a test database D'_i for the adjusted query Q'_i and the adjusted expected result R'_i by calling the RQP processor. The final test database D_i for the query Q_i and the query refinement F_i is created by taking the union of D'_i and DF_i; i.e., $D_i = D'_i \cup DF_i$.

For instance, in order to generate a test database D_1 for Q_1 and F_1 in the example above, MRQP first generates a test database D_F for the query refinement $F_1 = \{F_{11}\}$ and the expected results $RF_1 = \{RF_{11}\}$ as discussed in Section 2.2.1; e.g., a minimal test database DF_1 comprises one book with the given values (i.e., one book with the values specified for the attributes b_price, b_title and b_isbn by F_{11} and RF_{11}). Afterwards, the expected result R_1 is adjusted by executing $R'_1 = R_1 \ominus Q_1(DF_1) = \{< 900.0, 4 >\}$ and the query Q_1 is adjusted, too, which returns the adjusted query Q'_1:

```
Q1 :  SELECT SUM(b_price), COUNT(*)
      FROM book WHERE b_isbn!='0130402648'
```

Subsequently, we generate the test database D'_1 for the adjusted query Q'_1 and the adjusted result R'_1 (i.e., four books with the total sum \$900 that have an ISBN value other than '0130402648'). The final test database D_1 that returns R_1 for Q_1 and RF_{11} for F_{11} is created as the union of D'_1 and DF_1; i.e., $D_1 = D'_1 \cup DF_1$.

2.3 MRQP Procedure

The function *MRQP* which is shown in Figure 1 implements a general procedure for MRQP which generates a test database for a RQP-disjoint query set Q where some queries $Q_i \in Q$ can be recursively refined by a query refinement. The function *MRQP* first creates an empty database D for the query set Q (Line 1). Afterwards, the function checks for each query $Q_i \in Q$ if there exists a query refinement F_i for that query (Line 5). If yes, then this function generates a test database DF_i for that query refinement and the expected results RF_i by calling *MRQP* recursively (Line 6). Subsequently, the function adjusts the expected result R_i and the query Q_i w.r.t. DF_i (Line 7-8). Afterwards, the function *MRQP* creates the new test database D as a union of the existing test database D, the test database that is created for the adjusted query Q_i and the adjusted expected result R_i, and the test database DF_i generated for a potential query refinement F_i (Line 10). If the union does not satisfy the database schema S, then an error is returned (Line 11). If all queries in Q are processed the final test database D for Q and R is returned.

3. THE SPECIFICATION LANGUAGE MSQL

As discussed in the Section 2.2, we allow a tester to specify a test database which is adequate to execute a particular test case by manually creating a set of RQP-disjoint queries Q and a query refinement F_i for each query $Q_i \in Q$. In order to support the tester in formulating an RQP-disjoint query set Q and some query refinements, we have to decide whether Q is RQP-disjoint or not and whether a query refinement F_i refines a query $Q_i \in Q$ or not.

Consequently, in this section we define a database specification language called MSQL (based on SQL) and a Reverse Relational Algebra called MRRA which is used in MRQP to generate the test database (based on the Reverse Relational Algebra RRA of RQP in [4])[4]. For a query set Q and some query refinements for the queries $Q_i \in Q$ that are formulated in MSQL, we can easily check whether the query set Q is RQP-disjoint and if a query refinement F_i refines a query $Q_i \in Q$ (if the MRRA is used to reverse process these queries). However, we did not prove that there does not exist a more powerful language than MSQL that has the same properties.

Moreover, in this Section we illustrate an efficient solution for the *AdjustQuery* function and the *Adjust* operator $\ominus$ which are used to reverse process queries that are refined by a query refinement (as discussed in the section before) for all query classes of MSQL.

3.1 Query Classes and Algebra

In MSQL, a tester can formulate SQL `SELECT` queries with and without aggregations in the `SELECT` clause. Moreover, the queries supported by MSQL are not allowed to include join statements or subqueries and the predicate in the `WHERE` clause must be a conjunctive predicate in propositional logic that satisfies certain restrictions. More precisely, the supported query classes in MSQL are:

(1) Non-Aggregation queries which can be mapped to the following relational algebra expression:

$$\pi_A(\sigma_p(T))$$

where A represents the attributes and arithmetic functions in the `SELECT` clause, p is the selection predicate in the `WHERE` clause, and T is an arbitrary relation of the schema S.

(2) Aggregation queries which can be mapped to the following relational algebra expression:

$$\sigma_q(\chi_{B,COUNT(*)\ as\ c,AGG(D)}(\sigma_p(T)))$$

where q is the selection predicate in the `HAVING` clause, B represents the `GROUP-BY` attributes, $COUNT(*)$ is the non-distinct count function, $AGG(D)$ are the aggregation functions (AVG, MIN, MAX, SUM) in the `SELECT` clause on the attributes and arithmetic functions D, p is the selection predicate in the `WHERE` clause, and T is an arbitrary relation of the schema S.

The $COUNT(*)$ function is obligatory for aggregation queries (query class (2) above) because the *Adjust* operator $\ominus$ which is used in the *MRQP* function to process query refinements relies on that value (see Section 3.2). Moreover, for both query classes the selection predicate p must be a conjunctive predicate formulated in propositional logic. If a clause p_i in the conjunctive selection predicate p comprises an attribute a with a *primary-key* constraint, or a *unique* constraint, or a *foreign-key* constraint in the database schema S then p_i is only allowed to be a simple predicate expressing the equality of the attribute a and a constant value v (i.e., $a = v$).

[4]The RRA of RQP is the reverse variant of the relational algebra which pushes the expected query result from the root of a query tree down to the leaves in order to generate the test database.

The reverse relational algebra which is used to reverse process these query classes in MRQP is called MRRA. MRRA is similar to the RRA defined in [4]. The only difference is that the reverse selection operator (i.e., σ^{-1}) and the reverse join operator (i.e., $\bowtie^{-1}$) are not allowed to generate additional tuples that satisfy the negation of the selection predicate or the negation of the join predicate. Provided, that we use MRRA to generate a test database, then the following two theorems hold.

THEOREM 3.1. *Two arbitrary MSQL queries Q_j and Q_k are RQP-disjoint iff Q_j and Q_k specify tuples for different relations or $p_j \wedge p_k$ is not satisfiable which is decidable for the selection predicates in MSQL (p_j is the selection predicate representing the* `WHERE` *clause of Q_j and p_k is the selection predicate representing the* `WHERE` *clause of Q_k).*

PROOF (SKETCH) 3.2. *It is obvious that Q_j and Q_k are RQP-disjoint if Q_j and Q_k specify tuples in different relations because Q_k will be update independent from any* `INSERT` *statement which is generated by RQP for Q_j and an arbitrary expected result R_j of that query and vice versa. It immediately follows from [6] that Q_j and Q_k are RQP-disjoint if Q_j and Q_k read tuples from the same relation T and $p_j \wedge p_k$ is not satisfiable because all* `INSERT` *statements that could be generated for Q_j and an arbitrary expected result R_j by RQP satisfy p_j and thus will not be returned by Q_k which has the selection predicate p_k and vice versa.*

THEOREM 3.3. *An arbitrary MSQL query Q_j refines another arbitrary MSQL query Q_k iff the queries Q_j and Q_k read tuples from the same relation T and $(p_j \Rightarrow p_k)$ is valid which means that we have to show that $(!pj \vee pk)$ is valid or the negation $(pj \wedge !pk)$ is not satisfiable which is decidable for the selection predicates in MSQL (again, p_j is the selection predicate representing the* `WHERE` *clause of Q_j and p_k is the selection predicate representing the* `WHERE` *clause of Q_k).*

PROOF (SKETCH) 3.4. *It immediately follows from [6] that Q_j refines Q_k iff $(p_j \Rightarrow p_k)$ is valid, because all* `INSERT` *statements that could be generated for Q_j and an arbitrary expected result R_j by RQP satisfy p_j and thus will be returned by Q_k which has the selection predicate p_k.*

3.2 Adjust Operation for Query Refinements

The *AdjustQuery* function is used in the *MRQP* function (see Figure 1) to adjust the query Q_i so that calling RQP for Q_i does not generate any data for the expected result R_i which is returned by any query in the query refinement $F_i = \{F_{i1}, \ldots, F_{in}\}$ for the query Q_i. The implementation of this function is the same for both query classes of MSQL. We simply extract the selection predicate p_i of the query Q_i and the selection predicates p_{Fij} of each query $F_{ij} \in F_i$ and create a new selection predicate p_i' for the adjusted query Q_i' as $p_i' = p_i \wedge \neg p_{Fi1} \wedge \cdots \wedge \neg p_{Fin}$. An example for the *AdjustQuery* function was shown at the end of Section 2.2.2.

The *Adjust* operator $\ominus$ is used in the *MRQP* function (see Figure 1) to adjust the expected result R_i of a query Q_i w.r.t. the database DF_i generated for the query refinement F_i. The implementation of $\ominus$ for a non-aggregation query Q_i (query class (1) of MSQL) is a standard relational minus operator for bags. Additionally, the *Adjust* operator for non-aggregation queries checks if $Q_i(D_F) - R_i = \emptyset$ holds. Otherwise the expected result is not adjustable because the queries in the refinement F_i specify more tuples than the query Q_i which is not allowed by the definition. In that case the *Adjust* operator returns an error.

$\ominus$(**Relation** R, **Relation** S)
Output: Relation R'
```
(1)  R' = ∅ //Create an empty result
(2)  FOR EACH tuple r in R
(3)   r' = ∅ //Create empty tuple r'
(4)   //Extract tuple s from S
(5)   s = S.get(B, R.B)
(6)   IF(s==∅) r'=r
(7)   ELSE
(8)    if(B!=∅) r'(B) = r(B) //set B
(9)    r'(c) = r(c)-s(c) //set c
(10)  //Init results of agg.  functions
(11)  FOR EACH attribute agg(D) in AGG(D)
(12)   IF(agg==SUM)
(13)    r'(agg(D))=r(agg(D))-s(agg(D))
(14)   ELSE IF(agg==AVG)
(15)    r'(agg(D))=(r(agg(D))*r(c)-s(agg(D))*s(c))/r'(c)
(16)   ELSE IF(agg==MIN || agg==MAX)
(17)    IF(r(agg(D))!=s(agg(D))) r'(agg(D))=r(agg(D))
(18)  END FOR
(19)  END IF
(20)  R'.add(r') //add new tuple r' to R'
(21) END FOR
(22) RETURN R' //return result
```
Figure 2: Adjust operator $\ominus$

For an aggregation query Q_i (query class (2) of MSQL) the implementation of the *Adjust* operator is given in Figure 2. In that algorithm we refer to the group-by attributes B, the count value c, and the aggregation functions $AGG(D)$ that are defined by the expected result of an aggregation query. An example for that operator will be discussed in the next Section 3.3. The *Adjust* operator in Figure 2 is a binary operator which takes two relations R and S as input: R is the expected result of the query Q_i and S is the actual result of executing the query Q_i over the test database DF_i that was generated for the query refinement F_i (i.e., $S = Q_i(DF_i)$). The output of this operator is the adjusted result R'.

The implementation of the *Adjust* operator is as follows: The operator first creates an empty result R' (Line 1) and then iterates over all tuples in the expected result R (Line 2-21). For each tuple r in R an empty result tuple r' is created that should hold the adjusted values from r (Line 3). Afterwards, the tuple s is extracted from S that has the same values for the group-by attributes B in r. If query Q_i does not define a group-by attribute (i.e., $B = \emptyset$) then the only tuple in result S is returned (Line 5). If there does not exist such a tuple s, then the $\ominus$ operator uses r as the adjusted tuple r' and adds r' to R'. Otherwise, the $\ominus$ operator adjusts the expected query result (Line 7-19) as follows: First, the group by attributes B of r' are initialized with the attribute values $r(B)$ (Line 8). Then, the new count value is calculated as the difference of the original count value $r(c)$ and the count value $s(c)$ (Line 9). Finally, the adjusted expected results $r'(agg(D))$ for each aggregation function $agg(D) \in AGG(D)$ is created according to the type of the aggregation function (Line 10-18). Finally, the adjusted tuple r' is added to the result R' (Line 20). If all tuples r in R are processed the adjusted result R' is returned (Line 22). An example of that algorithm is given in the next subsection.

3.3 MSQL Example

Figure 3 gives a complete example of MRQP: Figure 3 (a) shows the database schema and Figure 3 (b) shows the RQP-disjoint query set Q and a query refinement F_1 which specify the test database for Test Case 5 of the online library (see Section 1). The query $Q_1 \in Q$ specifies the total number of all books in the test database. The other query $Q_2 \in Q$ specifies the password and the charges of a particular user with a certain password. The queries Q_1 and Q_2 are RQP-disjoint because they specify tuples for different tables. The

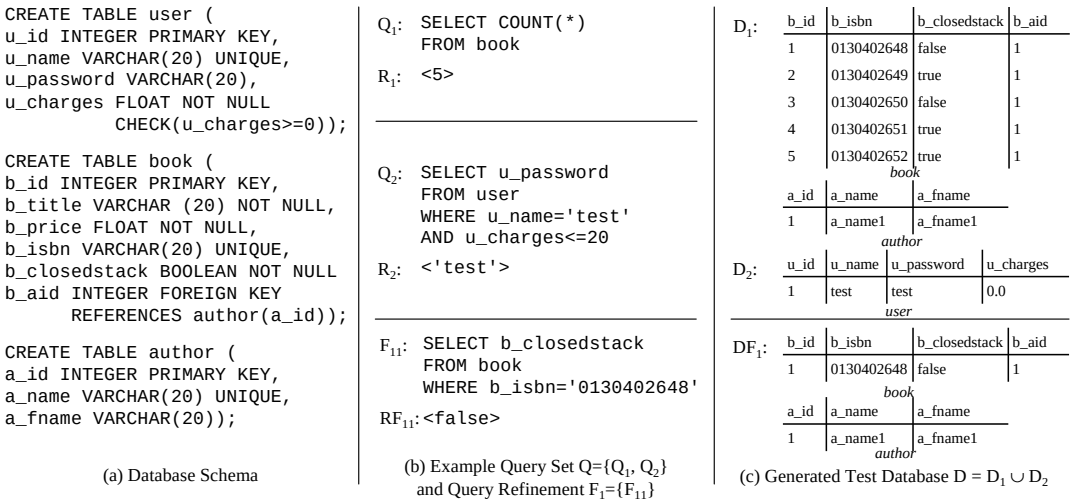

Figure 3: MSQL Example

query Q_1 is refined by a query refinement $F_1 = \{F_{11}\}$ and its expected results $RF_1 = \{RF_{11}\}$. The query F_{11} and the result R_{11} specify the value of the attribute $b\_closedstack$ of one book in the test database with a particular ISBN (i.e., $b\_isbn =$ '0130402648').

If we call the function *MRQP* in Figure 1 for the RQP-disjoint query set Q, then the test database DF_1 for the query refinement F_1 is generated first (DF_1 is shown in Figure 3 (c)). Due to the foreign-key handling in RQP [4] a tuple for the author with $a\_id = 1$ is created, too. As a next step, the adjusted expected result for query Q_1 is calculated as $R_1' = R_1 \ominus Q_1(DF_1)$: The query Q_1 is an aggregation query and $Q_1(DF_1)$ returns a count value of 1. According to the algorithm of $\ominus$ for aggregation queries (see Figure 2) the adjusted result R_1' has one tuple with the count value of 4. Afterwards, the query Q_1 is adjusted which results in Q_1':

```
Q1 : SELECT COUNT(*) FROM book
     WHERE b_isbn!='0130402648'
```

Afterwards, the test database D_1' for the adjusted query Q_1' and its adjusted expected result R_1' is generated which means that four tuples in the table *book* are created which satisfy $b\_isbn! =$ '0130402648' (D_1' is not shown separately in Figure 3 (c)). The database D_1 which is shown in Figure 3 (c) is the union of D_1' and the test database DF_1 generated for the query refinement F_1. Finally, the test database D_2 is generated for Q_2 and R_2 (Q_2 is not refined by a query refinement). As a last step, the final test database D is created as the union of D_1 and D_2 (i.e., $D = D_1 \cup D_2$).

4. RELATED WORK

In general, there has only been little related work which tackles the problem of generating test case aware databases (e.g., [12, 8]). However, these approaches do not directly apply for testing certain execution paths of an OLTP application (as MRQP does). In a technical point of view, [11] discusses a similar problem statement as RQP. However, only a very restricted set of relational expressions is supported in their work. Moreover, there has also been some work on efficient algorithms and frameworks to produce large amounts of test data for a given statistical distribution [9, 7]. That work is orthogonal to our work.

5. CONCLUSIONS AND FUTURE WORK

This work presented a new technique called MRQP which can be used to specify and generate test databases for OLTP applications. We have shown that MRQP is undecidable for arbitrary SQL queries. Consequently, we defined restrictions on the queries which can be used as input of MRQP so that MRQP can be solved efficiently. Moreover, we also defined a database specification language called MSQL (based on SQL) for which the user can easily check whether these restrictions are satisfied or not.

MRQP initially generates one test database per test case. However, in some cases it might be helpful to reduce the number of test databases. For this case we devised a trivial algorithm which merges individual test databases such that many test cases can use the same test database [5]. One avenue of future work is to find more efficient algorithms to reduce the total number of test databases which guarantee certain properties (e.g., minimality). Another avenue is to study the usability of MRQP in an industrial environment.

6. REFERENCES

[1] DTM Data Generator. http://www.sqledit.com/dg/.

[2] IBM DB2 Test Database Generator. http://www-306.ibm.com/software/data/db2imstools/db2tools/db2tdbg/.

[3] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[4] C. Binnig, D. Kossmann, and E. Lo. Reverse Query Processing. In *ICDE*, pages 506–515, 2007.

[5] C. Binnig, D. Kossmann, and E. Lo. Multi-Reverse Query Processing. Technical report, ETH Zurich, 2008.

[6] J. A. Blakeley, N. Coburn, and P. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. In *VLDB*, pages 457–466, 1986.

[7] N. Bruno and S. Chaudhuri. Flexible database generators. In *VLDB*, pages 1097–1107, 2005.

[8] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. An AGENDA for testing relational database applications. *Software Testing, Verification and Reliability*, 2004.

[9] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, pages 243–252, 1994.

[10] K. Houkjær, K. Torp, and R. Wind. Simple and realistic data generation. In *VLDB*, pages 1243–1246, 2006.

[11] T. Imielinski and J. Witold Lipski. Inverting relational expressions: a uniform and natural technique for various database problems. In *PODS*, pages 305–311, New York, NY, USA, 1983. ACM Press.

[12] A. Neufeld, G. Moerkotte, and P. C. Lockemann. Generating consistent test data for a variable set of general consistency constraints. *VLDB J.*, 2(2):173–213, 1993.

[13] J. M. Stephens and M. Poess. Mudd: a multi-dimensional data generator. In *WOSP*, pages 104–109, 2004.