Detecting Offline Transaction Concurrency Problems

Hao Luo[1], Mehedi Masud[2], and Hasan Ural[1]
[1]EECS, University of Ottawa, Ottawa, Canada
[2]Department of Computer Science, Taif University, Taif, Saudi Arabia
Email: {hao,ural}@site.uottawa.ca, mmasud@tu.edu.sa

Abstract— **When multiple instances of a database application (DBA) are executing concurrently, transactions from these instances interleave. Due to unintentional grouping of operations of the DBA into transactions, some interleavings of these transactions may cause offline concurrency problems. Although these problems are analogous to lost update, dirty read, non-repeatable read, and phantom problems, the concurrency controller of a DBMS cannot ensure ACID properties because from the DBMS's point of view the transactions involved in the offline concurrency problems are executed successfully. This paper gives the enumeration of patterns of interleaving of operations which identify potential offline concurrency problems in a DBA.**

Index Terms— **database concurrency, transaction processing, consistency, database application testing**

I. Introduction

A database application (DBA) is written in some source language where transactions are formed as sequences of SQL queries SELECT, INSERT, DELETE, UPDATE terminating with SQL keywords COMMIT or ROLLBACK. When multiple instances of a DBA are running, a set of transactions from these instances may be submitted to be executed by a Database Management System (DBMS). Since executing transactions in a given set one at a time seriously hampers the performance of the DBMS, transactions often are executed concurrently by interleaving operations (i.e., read, write, commit, abort as seen by the DBMS) from these transactions. However, allowing concurrent execution of transactions may create concurrency problems (Lost Update (P0), Dirty Read (P1), Non-repeatable Read (P2), and Phantom (P3)) [1]. Hence, DBMSs have been constructed to avoid the above problems, guarantee the ACID properties [1] for each transaction and accommodate different performance requirements via isolation levels (i.e., Read Uncommitted (0), Read committed (1), Repeatable Read (2), and Serializable (3) [1].

When transactions in a given set are executed at isolation level Serializable (which is by default), it is assumed that the transaction manager of the DBMS schedules the operations from these transactions in such a way that the result of the concurrent execution of the operations is the same as that of a sequential execution of the transactions in some order. It is also assumed that none of the concurrency problems P0 to P3 could occur, provided that the DBA developer uses transaction constructs appropriately. However, this last assumption may not hold if an unintentional transaction design/implementation in the DBA groups SQL queries into transactions such that data are manipulated across multiple transactions when all the work should be done in a single transaction. Potential problems analogous to P0 to P3 that may occur due to this unintentional grouping of operations are called offline concurrency problems [2]. The DBMS will not be aware of the existence of such problems because none of the ACID properties has been violated and therefore concurrency control mechanisms in the DBMS cannot prevent realization of offline concurrency problems.

The offline concurrency problems in a DBA are identified by [3], [4] by considering three transactions, T_i, T_j, and T_k, from two concurrently executing instances of the DBA and by analyzing the dataflow involving attributes of a relational database and host variables between transactions at the highest isolation level. They provided two tables indicating patterns of operations on related attributes/host variables in T_i, T_j, T_k which may pose potential offline concurrency problems. However, justifications for the correctness and completeness of the identified potential offline concurrency problems have not been provided. Also, they have not considered other isolation levels. In this paper, we complete their analysis. In the following sections, we identify potential offline concurrency problems in addition to the problems identified by [3], [4]. We demonstrate the existence of these additional potential offline concurrency problems we identify by examples. We provide justifications for the claims for the absence of each potential offline concurrency problem for each pattern. We also enumerate the problematic patterns in other isolation levels.

II. Identifying Potential Offline Concurrency Problems in DBAs

In a transaction, SQL queries are formed using attributes and host variables. A READ is an occurrence of an attribute V in an SQL query by which the value of V is accessed. A WRITE is an occurrence of an attribute V in an SQL query by which a value is assigned to V. A DEF is an occurrence of a host variable v in an SQL query by which a value is assigned to v. A USE is an occurrence of a host variable v in an SQL query by which the value of v is referenced. The notions of DEF

doi:10.4304/jsw.7.8.1855-1860

TABLE I.
PROBLEMATIC PATTERNS SET 1

	Patterns							
	1	2	3	4	5	6	7	8
T_i	R	R	R	R	W	W	W	W
T_j	R	R	W	W	R	R	W	W
T_k	R	W	R	W	R	W	R	W
$Concurrency\ problems$	No	No	Yes P2 **P3**	Yes P0	No	Yes P1	Yes P0 P3 **P2**	Yes P0 **P1**

TABLE II.
PROBLEMATIC PATTERNS SET 2

	Patterns							
	9	10	11	12	13	14	15	16
T_i	U	U	U	U	D	D	D	D
T_j	R	R	W	W	R	R	W	W
T_k	U	D	U	D	U	D	U	D
$Concurrency\ problems$	**Yes P1**	**Yes P1**	Yes P0 P1 P2 P3	Yes P0 P1 P2 P3	**Yes P1**	**Yes P1**	Yes P0 P1 P2 P3	Yes P2 P3

and USE are generally used in data flow analysis [5]–[8] to form def-use associations [9]. Data flow analysis is a technique for optimizing programs by compilers [10] or determining the possible set of values of variables at various points in a program for validation purposes [11]. We use the following convention to classify the variable (attribute or host variable) occurrences appearing in SQL queries (SELECT, INSERT, UPDATE, DELETE) as READ, WRITE, DEF, and USE. Consider a database containing a relation r with attributes X and Y. Let x and y be some host variables in the source code of a DBA.

- SELECT X INTO x FROM r WHERE Y = y, maps to READ of X,Y; DEF of x and USE of y.
- INSERT INTO r VALUES(x), maps to WRITE of X; USE of x.
- UPDATE r SET X = f(x) WHERE Y = y, maps to WRITE of X and READ of Y; USE of x, y.
- DELETE FROM r WHERE X = x, maps to READ of X; USE of x.

Note that an attribute or a host variable is said to be $related$ to a host variable or an attribute, respectively, if the value of one is determined with respect to the other. If the value of a variable $z1$ (attribute or host variable) determines the value of other variable $z2$, then we say that $z1$ *affects* $z2$.

The problem studied in this paper is stated as follows: Consider a DBA and its two instances A and B which are running concurrently. Suppose that a database containing a relation r is used by this DBA. Further suppose that transactions T_i and T_k belong to instance A and transaction T_j belongs to instance B. Let T_i, T_j, and T_k be executed in this order to represent the interleaving of transactions of instances A and B. Assume that T_j accesses some attribute(s) of relation r that is (are) accessed by T_i and T_k; or T_j accesses some attribute of a relation r that affects some host variable(s) defined/used in T_i and T_k. Identify patterns of READ, WRITE, DEF, USE of related attributes/host variables in T_i, T_j, T_k which may pose potential offline concurrency problems.

There are two essential characteristics for a pattern of READ, WRITE, DEF, USE to indicate an offline concurrency problem. First, the pattern should result in different values for an attribute in two instances of a DBA. Therefore, we should examine sixteen patterns shown in Tables I and II where each column represents a pattern which indicates whether T_i, T_j, T_k has a READ (R), WRITE (W), DEF (D), or USE (U) of related attributes/host variables. Second, either instance A or instance B should use its value of this attribute later in its execution and create a scenario that is described by any of P0, P1, P2 and P3. Only when both of the above two requirements are satisfied, we say that the pattern indicates an offline concurrency problem. The potential offline concurrency problems associated with each pattern are shown at the last row of Tables I and II where problems identified by Deng et al. [3], [4] are in regular font. Our analysis of the sixteen patterns resulting from sixteen combinations of R, W, D, U show that the problems identified by Deng et al. are correct but they are not complete. There are additional potential offline concurrency problems associated with some of these patterns which are shown in bold in Tables I and II.

A. Examples for Additional Offline Concurrency Problems

We show below examples of the additional offline concurrency problems we identified. We use C programming language with embedded SQL queries in the examples. In all of these examples, we assume that two instances of a DBA, instance A (executing transactions $T1$ and $T2$)) and instance B (executing transaction $T3$) are running concurrently. We also assume that the transactions $T1$, $T2$, $T3$ are executed in the order of $T1(A), T3(B), T2(A)$.

Example 1 (Pattern 3, Problem P3):
Consider a database with a single relation staff(Employee_ID, Position, Salary), denoted by $r(X, Y, Z)$, and a DBA which contains the following:

ChangeSalary(y, a) which first changes the salaries of employees whose position is y by a percentage a and then for employees with the position y, reports their total salary, their average salary and the standard deviation of their salaries.

AddStaff(x, y, z) which inserts a new employee into r.

The details of ChangeSalary(y, a) and AddStaff(x, y, z) are given below.

ChangeSalary(y, a)
{ int x, z;
Transaction T1:
UPDATE r SET Z=:Z*(1+a) WHERE Y=:y
SELECT SUM(Z) INTO :z FROM r WHERE Y=:y
printf("Employees of position, $\%s$, make a total of \$, $\%d$ after salary changes", y, z); commit
Transaction T2:
SELECT COUNT(X) INTO :x FROM r WHERE Y=:y
printf("Number of employees of position, $\%s$ is $\%d$ and they make \$, $\%d$ on average ",y, x, z/x);
SELECT STDEV(Z) INTO :z FROM r WHERE Y=:y
printf("The standard deviation of the salaries is $\%d$",z); commit }
AddStaff(:x, :y, :z)
{ Transaction T3:
INSERT INTO r VALUES(:x, :y, :z)
commit }

Table III shows how the problem P3 is generated considering the values of the attribute Z in r and when the value of y is the same in both A and B.

TABLE III.
EXAMPLE OF PROBLEM P3 IN PATTERN 3

Transactions	Operation	Description
T1	R(Z)	increases salaries of employees
T3	W(Z)	adds a new employee with position y
T2	R(Z)	computes average salary

Observe from Table III that the problem P3 in Pattern 3 (R W R) occurs. The reason is that the newly added employee with position y is not counted in the sum of employees salaries whose position is y, but is counted in the standard deviation of employees salaries whose position is y.

Example 2 (Pattern 7, Problem P2):
Consider a database that contains a single relation flights(flight_num, vacancy, price), denoted by $r(W, Y, Z)$, and a DBA which contains the following:

CancelBooking(w) which cancels a booking of a flight w, increases the number of vacancies of the flight w by 1 and shows the number of seats available.

Reserve(w) which reserves a booking in a flight w and decreases the number of vacancies of the flight w by 1.

The details of CancelBooking(w) and Reserve(w) are as follows:

CancelBooking(w)
{ int y;
Transaction T1:
SELECT Y INTO :y FROM r WHERE W=:w
UPDATE r SET Y=:y+1 WHERE W=:w
commit
Transaction T2:
SELECT Y INTO :y FROM r WHERE W=:w
printf("Flight $\%$d now has $\%d$ seats available for booking", w,y); commit }
Reserve(w)
{ int y;
Transaction T3:
SELECT Y INTO :y FROM r WHERE W=:w
if(y>0) { UPDATE r SET Y=:y-1 WHERE W=:w
commit } }

Now assume that in r there are two seats available before of the execution of the transactions. Table IV shows how the problem P2 is generated considering the values of the attribute Y in r. The table also shows the values of Y and the host variable y during the execution of the transactions.

Observe from Table IV that the problem P2 in Pattern 7 (W W R) occurs. The reason is that after A updates Y there are 3 seats available for the flight w but later A reads and outputs Y which shows that there are only 2 available seats. This happens because B updates the value of Y before A reads Y again.

Example 3 (Pattern 8, Problem P1):
Consider the relation flights(flight_num, vacancy, price) in the previous example and a DBA which contains the following:

SetPromotion(w) which sets the ticket price of a flight w to 80% of the original price if the flight has more than 20 vacancies and also ensures that no seat has a price lower than \$195.

Surcharge(w) which adds a new charge of \$30 to the price of a seat for the flight w.

The details of SetPromotion(w) and Surcharge(w) are given below.

SetPromotion(w)
{ int z;
Transaction T1:
UPDATE r SET Z=:Z*.08 WHERE W=:w
SELECT Z INTO :z FROM r WHERE W=:w
SELECT Y INTO :y FROM r WHERE W=:w AND Z=:z
if(y<20) rollback else commit
Transaction T2:
if(z<195) UPDATE r SET Z=195 WHERE W=:w
commit }
Surcharge(w)
{ Transaction T3: UPDATE r SET Z=Z+30 WHERE W=:w
commit }

Now assume that in r there are 21 seats available for the flight w and the price of the ticket is \$200 before the execution of the transactions. Table V shows how the problem P1 is generated considering the values of the attribute Z in r. The table also shows the values of Z and the host variables y and z during the execution of the transactions.

Observe from Table V that the problem P1 in Pattern 8 (W W W) occurs. The reason is that B assigns a value

TABLE IV.
EXAMPLE OF PROBLEM P2 IN PATTERN 7

Transaction	Operation	Read Value (Y)	Value in y	Updated Value (Y)
T1	W	2	2	3
T3	W	3	3	2
T2	R	2	2	2

TABLE V.
EXAMPLE OF PROBLEM P1 IN PATTERN 8

Transaction	Operation	Read Value (Z)	Value of z, y	Updated Value (Z)
T1	W	200	160, 21	160
T3	W	160		190
T2	W	160	160,21	195

to Z which does not exist due the later update of Z by the transaction $T2$ in A.

The remaining four additional patterns that are associated with the problem P1 are the same as the problem P1 associated with Pattern 6 (W, R, W) because all the Ds and Us in these four patterns must indeed be DEFs and USEs of those host variables that are related to the attribute that is written in order to potentially cause the problem P1. Thus, these four patterns are the same as Pattern 6. Hence we give an example for the problem P1 exhibited by this pattern. Examples for the problem P1 for the four additional patterns can easily be constructed by variations of this example.

Example 4 (Pattern 6, Problem P1):
Consider a DBA which contains the following: a new version of SetPromotion(w) of Example 3, (called SP2 below), which sets the price of flight w to 80% of the original price if it has more than 20 vacancies, but also makes sure the price is not lower than 195 and Quote(w) which shows the price for a specified flight w.

The details of SetPromotion(w) and Surcharge(w) are given below.

SetPromotion(w) //SP2
{ int z;
Transaction T1:
SELECT Z INTO :z FROM r WHERE W = :w
UPDATE r SET Z = :z*0.8 WHERE W = :w AND Y $>$ 20
commit
Transaction T2:
SELECT Z INTO :z FROM r WHERE W = :w
if (z $<$ 195) UPDATE r SET Z = 195 WHERE W = :w
commit }
Quote(w)
{ int z;
Transaction T3:
SELECT Z INTO :z FROM r WHERE W = :w
printf("The price for flight %d, is, %d", w, z); commit
}

Now assume that in r there are 21 seats available for the flight w and the price of the ticket is \$200 before the execution of the transactions. Table VI shows how the problem P2 is generated considering the values of the attribute Z in r. The table also shows the values of Z and the host variable z during the execution of the transactions.

Observe from Table VI that the problem P1 in Pattern 6 (W R W) occurs since B reads a value of Z that does not exist in the database.

B. Justification of Non-problematic Patterns

From Tables I and II we observe that there are patterns that do not create any concurrency problems, and some patterns only create specific subsets of problems P0, P1, P2, P3. Below, we show why these patterns cannot lead to any or some potential offline concurrency problems.

Pattern 1: All three occurrence types in T_i, T_j, T_k are READ. The value of the attribute read by these transitions will not be changed by any of these transactions, thus its value will be kept consistent among instances A and B. Therefore we will not have any concurrency problems.

Pattern 2: The last occurrence type is WRITE, which changes the value of the attribute read by T_i and T_j. T_j in instance B will have an old value of this attribute after T_k changes the value of the attribute. There will not be any concurrency problem unless instance B uses this value later in its execution. But even then, that usage together with this READ in T_j will be caught by some other pattern. Therefore this pattern will not have any concurrency problems.

Pattern 3: This pattern will result in two different values of the same attribute in instance A and only one of these two values is consistent with the value of the same attribute in instance B. It is a typical description of P2 or P3. Both P0 and P1 require T_k to change the value of the attribute read in T_i, thus cannot occur for this pattern. Therefore this pattern cannot lead to concurrency problems P0 and P1.

Pattern 4: This pattern will result in using the value of an attribute by instance A that is read before instance B changes the value of this attribute. It is a typical description of P0. P1 cannot happen because T_i does not change the value of the attribute in instance A. P2 and P3 need T_k in instance A to read the value of the attribute again in order to have a different value than first read in T_i in instance A. Thus they cannot occur for this pattern. Therefore this pattern cannot lead to concurrency problems P1, P2 and P3.

Pattern 5: The first occurrence type is WRITE in T_i of instance A, which changes the value of the attribute. T_j

TABLE VI.
EXAMPLE OF PROBLEM P1 IN PATTERN 6

Transaction	Operation	Read Value (Z)	Value of z	Updated Value (Z)
T1	W	200	200	160
T3	R	160	160	160
T2	W	160	160	195

in instance B reads this attribute after its value is changed by instance A. Then, T_k in instance A reads the attribute. The attribute's value will be the same for both instances A and B after T_i, T_j, T_k are executed. Therefore, we will not have any concurrency problems.

Pattern 6: In this pattern, instance A changes the value of an attribute twice. P0, P2 and P3 all require instance B to change the value of this attribute, which does not happen in the pattern. Therefore this pattern cannot lead to concurrency problems P0, P2 and P3.

Pattern 7: Instance A changes the value of an attribute before instance B changes the value of the same attribute. Instance A does not attempt to change the value of this attribute later thus contradicts to definition of P1. Therefore this pattern cannot lead to concurrency problem P1.

Pattern 8: P2 and P3 are caused by the difference of two READs of an attribute in one instance. In this pattern, instance A does two WRITEs instead. Therefore this pattern cannot lead to concurrency problems P2 and P3.

Patterns 9, 10, 13 and 14: All concurrency problem definitions except that of P1 require instance B to change the value of the attribute in T_j. Therefore this pattern cannot lead to concurrency problems P0, P2 and P3.

Pattern 16: P0, P1 are not possible because instance A redefines its variable in T_k using the value of the attribute that is changed by instance B. Even if instance A changes the value of this attribute later, the value it will be using is based on the changes made in instance B. Therefore this pattern cannot lead to concurrency problems P0 and P1.

C. Problem Patterns in Other Isolation levels

We also observe that all the patterns at SERIALIZABLE level also be problematic at lower isolation levels (Read Uncommitted, Read Committed, and Repeatable Read). However, from SERIALIZABLE downwards, at each isolation level there is one additional offline concurrency problem from P1 to P3. Tables VII, VIII, and IX show the additional problematic patterns at different isolation levels. Note that the problem P0 is not possible at any of the isolation levels from 0 to 2 [12]. Therefore, all occurrences of P0 in Table VIII and Table IX refer to the offline concurrency problem P0 as discussed in Section II-B. In this case we consider only two transactions T_i and T_j.

TABLE VII.
PROBLEM PATTERNS AT ISOLATION LEVEL 2

T_i	R	U	U	D	D
T_j	W	W	W	W	W
T_i	R	U	D	U	D
$Concurrency\ problems$	Yes	Yes	Yes	Yes	Yes
	P3				

TABLE VIII.
PROBLEM PATTERNS AT ISOLATION LEVEL 1

T_i	R	R	U	U	D	D
T_j	W	W	W	W	W	W
T_i	R	W	U	D	U	D
$Concurrency\ problems$	Yes	Yes	Yes	Yes	Yes	Yes
	P2	P0	P2			

III. RELATED WORK

There is significant research work focusing on testing and analysis of programs, however, less attention is given specifically to examine and detect offline concurrency problems for database-centric applications. Related work on database-centric application testing use control flow and data flow aspects of the database application to be tested. Chan and Cheung propose a technique that tests database applications that are written in a general purpose programming languages, such as Java, C, or C++, and include embedded structured query language statements that are designed to interact with a relational database [13], [14]. However, their focus on the control flow of a program ignores important information about the flow of data between the program and the databases. Daou et al. use data flow information to support the regression testing of database-centric applications [15]. However, their use of data flow analysis does not consider either a representation for a database-centric application or a complete description of a database interaction association. Data flow coverage measures are also used to determine the adequacy of SQL select queries in light of a database that has already been populated with data [16].

Chays et al. discuss several problems associated with testing database applications and propose the AGENDA tool as a solution to some of these problems [17]–[19] . In [17], the authors propose a partially automated software testing technique, inspired by the category-partition method [20], that attempts to determine if a program behaves according to its specification. In [18], Chays and Deng have extended AGENDA to support testing of database transaction concurrency by using a data flow analysis to determine database transaction schedules that may reveal program faults.

Neufeld et al. propose techniques that are similar to [17]–[19] because they generate database states using knowledge of the constraints in the relational schema [21], [22]. However, neither of these approaches explicitly

TABLE IX.
PROBLEM PATTERNS AT ISOLATION LEVEL 0

T_i	W	R	W	W	U	U	D	D
T_j	R	W	W	R	R	R	R	R
T_i	R	W	W	W	U	D	U	D
$Concurrency$	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
$problems$	P1	P0		P1				

provides a framework to support offline concurrency issues of database applications.

IV. CONCLUSION AND FUTURE WORK

In this paper we have identified additional offline concurrency problems in the SERIALIZABLE isolation level and justified why some patterns do not create offline concurrency problems and why some patterns create only some subset of {P0, P1, P2, P3}. We have also shown the problematic patterns that can occur in other isolation levels (i.e., READ UNCOMMITTED, READ COMMITTED, and REPEATABLE READ).

The algorithm given by [18], [19], [23] can be augmented to detect the additional problematic patterns we have identified. One drawback of this algorithm is that it is based on a restricted notion of influence, which is identical to the definition-use association [9]. This restricts the identification of potential offline concurrency problems to those instances which are associated with a pair of elements forming the definition-use association. This algorithm needs to be generalized to identify potential offline concurrency problems which are associated with elements on a chain of definition-use associations.

Like potential offline concurrency problems, there can be potential offline recoverability problems. Our ongoing work considers the questions of how and why the patterns can cause potential offline recoverability problems.

REFERENCES

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency control and recovery in database systems," *Addison Wesley, Reading*, 1987.

[2] M. Fowler, *Patterns of enterprise application architecture*. Addison Wesley and Benjamin Cummings, 1987.

[3] Y. Deng, P. Frankl, and Z. Chen, "Testing database transaction concurrency," in *Proc. of the 18th IEEE International Conference on Automated Software Engineering*, 2003.

[4] Y. Deng, "Testing database transactions," Ph.D. dissertation, Department of Computer Science, Polytechnic University in Brooklyn, USA, 2005.

[5] E. Frances and J. Cocke, "Graph theoretic constructs for program flow analysis," T.J. Watson Research Center, Yorktown Heights, Tech. Rep., 1972.

[6] E. Weyuker, "The complexity of data flow criteria for test data selection," *Information Processing Letters (IPL)*, vol. 19, no. 3, pp. 103–109, 1984.

[7] G. Fraser and F. Wotawa, "Ordering coverage goals in model checker based testing," in *Proc. of the ICST*, 2008, pp. 31–40.

[8] K. Kennedy and L. Zucconi, "Applications of a graph grammar for program control flow analysis," in *Proc. of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming languages*, 1977, pp. 72–85.

[9] S. Rapps and E. Weyuker, "Data flow analysis techniques for test data selection," in *Proc. of the IEEE International Conference on Software Engineering*, 1982.

[10] A. Aho, R. Sethi, and J. Ullman, *Compiler: Priciples, Techniques, and Tools*. Addison Wesley, 2007.

[11] H. Ural, K. Saleh, and A. Williams, "Test generation based on control and data dependencies within system specifications in sdl," *Computer Communications*, vol. 23, no. 7, pp. 609–627, 2000.

[12] H. Berenson, "A critique of ansi sql isolation levels," in *Proc. of the ACM Special Interest Group on Management of Data Conference*, 1995.

[13] M. Chan and S. Cheung, "Applying white box testing to database applications," Hong Kong University of Science and Technology, Department of Computer Science, Tech. Rep., 1999.

[14] ——, "Testing database applications with sql semantics," in *Proc. of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, 1999.

[15] B. Daou, R. Haraty, and N. Mansour, "Regression testing of database applications," in *Proc. of the ACM Symposium on Applied Computing*, 2001.

[16] M. Suarez-Cabal and J. Tuya, "Using an sql coverage measurement for testing database applications," in *Proc. of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2004.

[17] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weyuker, "A framework for testing database," in *Proc. of the 7th International Symposium on Software Testing and Analysis*, 2000.

[18] D. Chays and Y. Deng, "Demonstration of agenda tool set for testing relational database applications," in *Proc. of the International Conference on Software Engineering*, 2003.

[19] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker, "Agenda: A test generator for relational database applications," Department of Computer and Information Sciences, Polytechnic University, Brooklyn, NY, Tech. Rep., 2002.

[20] T. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[21] A. Neufeld, G. Moerkotte, , and P. Lockemann, "Generating consistent test data: Restricting the search space by a generator formula," *VLDB Journal*, vol. 2, pp. 173–213, 1993.

[22] J. Zhang, C. Xu, and S. Cheung, "Automatic generation of database instances for whitebox testing," in *Proc. of the 25th Annual International Computer Software and Applications Conference*, 2001.

[23] D. Chays, Y. Deng, P. Frankl, S. Dan, F. Vokolos, and E. Weyuker, "An agenda for testing relational database applications," *Software Testing, Verification, and Reliability*, vol. 14, no. 1, pp. 17–44, 2004.