

Testing Database Transaction Consistency

Yuetang Deng
Phyllis Frankl
Polytechnic University
Six Metrotech Center
Brooklyn NY 11201 *
{ytdeng, pfrankl}@cis.poly.edu

David Chays
Adelphi University
1 South Avenue
Garden City, NY 11530
chays@adelphi.edu

Abstract

AGENDA is a tool set for testing relational database applications. In this paper we extend AGENDA to test transaction consistency. Two levels of checks are used to check both database state and state transition. The transition check validates the state transition performed by the transaction. The state check validates that the overall global consistency properties hold for the new database state. Our tool set can handle general SQL assertions that are defined in the SQL standard but are not supported by current DBMSs, such as constraints involving multiple tables and SQL aggregation functions. A test generation heuristic that takes account of certain constraints is also presented. A case study based on the TPC-C benchmark shows promising results.

1. Introduction

Database and transaction processing systems occupy a central position in our information-based society. It is essential that these systems function correctly and provide acceptable performance. Substantial effort has been devoted to assuring that the algorithms and data structures used by Database Management Systems (DBMSs) work efficiently and protect the integrity of the data. However, relatively little attention has been given to developing systematic techniques for assuring the correctness of the related application programs. Given the critical role these systems play in modern society, there is clearly a need for new approaches to assessing the quality of the database application programs.

To address this need, we have developed a systematic, partially-automatable approach to testing database applications and a tool set, AGENDA, based on this approach. AGENDA has tools for populating the database with data satisfying integrity constraints expressed in the schema, for

generating inputs to the application and for checking the output and resulting database state. In initial work, described in [2], we identified challenges in testing database applications, proposed an approach to testing and a toolset to support it, and implemented an initial version of one of the tools, the state generator. We later redesigned the tool set so that it used a database, rather than a complex in-memory data structure, to store relevant information about the program under test and to facilitate communication between tools. We implemented preliminary versions of tools for generating inputs to the application, and checking outputs and database state after executing the application. That work is reported in [3].

Our earlier work focused on applications consisting of individual queries and placed limitations on the kinds of integrity constraints handled. That initial work served as a proof of concept and provided understanding of key issues. However, most real database applications are not single queries; rather, they consist of multiple queries embedded in a host program. Related queries are grouped into transactions, which may have complex semantic consistency constraints, describing relationships that are expected to always hold among the database elements and the way the transaction is intended to transform the database state. In this paper we explore problems dealing with testing transactions, focussing on whether transactions are consistent. The topic of concurrency problems associated with multiple clients simultaneously running an application program is addressed in [4].

We consider relational database applications. A relational database (DB) consists of a collection of interrelated tables. The structure of the DB is described by a *schema* written in the SQL data definition language. In addition to specifying the attributes (column names and associated data types) of each table, the schema may specify *integrity constraints* that the data must satisfy. These may include not-NULL constraints, indicating that a particular attribute must have a value; domain constraints, restricting the set

*Supported in part by NSF Grant CCR-9988354

of values for an attribute; uniqueness or key constraints, indicating that no two rows may have the same value for a given attribute or collection of attributes; referential integrity or foreign key constraints, indicating that one or more attributes from a row in one table must also appear in a row of another table; or complex semantic constraints restricting the relationship among values of attributes. The database management system (DBMS) translates these high level descriptions of data into low-level data storage, access and checking primitives, freeing the programmer of the burden of worrying about such details.

A database application program performs queries that may retrieve data from the DB, modify data, and insert or delete rows of tables. We consider applications in which queries are embedded into a host program, written in a high-level language, such as Java or C++. These queries may contain *host variables*, i.e., variables of the host program. In order to protect the DB from corruption due to a variety of causes, including hardware failure, interference from concurrently executing applications, and, to some extent, bad data values, the application programmer can group related queries into *transactions*. The DBMS assures that all of the queries in a transaction can execute successfully and that the resulting DB state satisfies the integrity constraints before allowing the transaction to *commit*, permanently saving the new DB state. If a transaction cannot *commit*, it fails.

This paper focuses on testing transactions to ensure that they are *consistent* with their requirements. This entails populating the DB, supplying values for all host variables that serve as inputs to the transaction, and checking the results. The main contribution of the paper is a technique for automatically checking results. Our technique automatically generates test oracles (based on constraints in the schema and on preconditions, postconditions, and constraints supplied by the tester) and automatically executes these oracles. The main idea of this technique is that complex constraints supplied by the tester are automatically translated into simpler constraints that are automatically enforced by the DBMS transaction manager. A preliminary version of this technique was described in [3]; it is substantially extended here to handle much more complicated constraints. In addition, we present improved techniques for generating input values for the host variables. A case study based on the TPC-C benchmark for transaction processing is presented.

In general, an *integrity constraint* or *state constraint* is a predicate over database states. It is intended to be true in all legitimate database states. These are sometimes called *global consistency constraints* because they may involve many database tables. A database is said to be in a consistent state if all of its constraints are true in that state. Not all database states are allowed. There are two reasons for this:

Internal consistency. It is often convenient to store the same information in different forms. For example, the TPC-C specification defines a total of twelve consistency constraints, one of which says that the sum of certain attributes in one table should be equal to an attribute in another table.

Business rules. Business rules restrict the possible states of the enterprise. When such a rule exists, the possible states of the database are similarly restricted. For example, in a student registration system, the number of students registered for a course must not exceed another number stored in the database, the maximum enrollment for that course. A state in which the number of registrants is greater than the maximum enrollment is not allowed.

Database designers typically define various constraints that all database states should satisfy. These include constraints to insure that the values of attributes are sensible (e.g. not NULL constraints, domain constraints), to assure that certain attribute values (or combinations) appear only once in a table (uniqueness constraints), and to assure that data in related tables are consistent with one another (foreign key constraints). SQL provides syntax for expressing such constraints as part of the database schema and most DBMSs can enforce them by checking for violations when the database state is modified. In addition there may be complex semantic constraints intended to insure that the database state accurately reflects the real-world entity that it is modeling and that it satisfies business rules of the organization. Some of these semantic constraints may be explicitly expressed in the schema (via SQL declarations) and some may be implicit constraints, expressed only in supplementary documentation either in SQL or in natural language. Some of the semantic constraints that are explicitly included in the schema may be enforced by the DBMS, but others will not be. To be consistent, the database state must satisfy all integrity constraints. A database is *consistent* if all integrity constraints are satisfied.

Transaction consistency has two aspects: when run in isolation starting from a consistent database state, a transaction should produce a new database state that is also consistent; furthermore, the relationship between the new state and the old state should satisfy particular requirements of the transaction's specifications. Producing consistent transactions is the responsibility of the application programmer [6]. Consequently, it is error prone.

We consider global database constraints that can be expressed in SQL. Some of these are simple constraints that can be expressed with SQL's not NULL, uniqueness, and foreign key constraints and enforced by the DBMS. Others are complex constraints describing restrictions on the real world entity that the data is modeling or describing "business rules" of the enterprise. Section 3 of this paper describes our technique for checking global consistency during testing. It can handle complicated constraints involving

multiple tables and SQL aggregation functions.

In addition to transforming consistent states to consistent states, each transaction must satisfy its own requirements specification. We express these with assertions relating the database state before execution of the transaction to the expected state after execution. Section 4 describes a mechanism to log changes to the database state and check that they satisfy the requirements.

The initial AGENDA prototype took account of not NULL, uniqueness, and referential integrity constraints in generating an initial database state, but not of semantic constraints. Section 5 shows how the initial state can be updated to create a state satisfying more complex constraints. Complex constraints, along with interactions between different queries in a transaction, also make the task of generating test inputs more difficult. More refined heuristics for input generation are also discussed in section 5.

All of these techniques have been implemented in the current version of AGENDA. Section 6 describes experience using AGENDA on a substantial realistic example, an implementation of the TPC-C benchmark with seeded errors.

2. Background

2.1. Agenda Tool Set

In our previous work [2, 3], we have discussed issues arising in testing database systems, presented an approach to testing database applications, and described AGENDA, a set of tools based on this approach. In testing such applications, the states of the database before and after application's execution play an important role, along with the user's input and the system output. A framework for testing database applications was introduced. A complete tool set based on the framework was prototyped. The components of this system are: Agenda Parser, State Generator, Input Generator, State Validator, and Output Validator.

AGENDA takes as input the database schema of the database on which the application runs; the application source code; and "sample value files", containing some suggested values for attributes. The tester interactively selects test heuristics and provides information about expected behaviors of test cases. Using this information, AGENDA populates the database, generates inputs to the application, executes the application on those inputs and checks some aspects of correctness of the resulting database state and the application output. This approach is loosely based on the Category-Partition method: the user supplies suggested values for attributes, partitioned into groups, which we call data groups. This data is provided in the sample value files. The tool then produces meaningful combinations of these

values in order to fill database tables and provide input parameters to the application program. Data groups are used to distinguish values that are expected to result in different application behaviors. For example, in a payroll system, different categories of employees may be treated differently. Additional information about data groups, such as probability for selecting a value from the list of values that follows, can also be provided via sample value files.

Using these data groups and guided by heuristics selected by the tester, AGENDA produces a collection of *test templates* representing abstract test cases. The tester then provides information about the expected behavior of the application on tests represented by each test template. For example, the tester might specify that the application should increase the salaries of employees in the "faculty" group by 10% and should not change any other attributes. In order to control the explosion in the number of test templates and to force the generation of particular kinds of templates, the tester selects heuristics. Finally, AGENDA instantiates the templates with specific test values, executes the test cases and checks that the output and new database state are consistent with the expected behavior indicated by the tester.

Although the initial AGENDA prototype could accept any SQL schema and any single query, it did not take account of semantic constraints when populating the database. In checking the state after execution of the query, it could only check relatively simple properties.

In our enhanced version of AGENDA, the testing process is as follows:

1. A source-level tool constructs the Control Flow Graph from the application source code, groups the queries into transactions according to BEGIN/END TRANSACTION and COMMIT/ROLLBACK statements..
2. After the Agenda Parser parses the application schema, the State Validator generates a log table for each application table and generates rules/triggers to update the log tables automatically.
3. After the Agenda Parser parses the application query, the tester selects the transaction to be tested, and AGENDA generates all the test templates for the transaction.
4. The State Generator generates an initial application DB, guided by the integrity constraints in the schema and heuristics selected by the tester.
5. The tester has the option to define global consistency properties, and AGENDA updates the initial DB state to satisfy these properties.
6. AGENDA generates inputs based on the heuristics chosen by the tester.

7. For each transaction, the tester specifies the preconditions/post-conditions.
8. AGENDA runs the application on the generated test cases, checks the changes stored in the log tables against the transaction constraints, checks the new application DB state against the global consistency constraints, and reports the results.

2.2. TPC-C Benchmark

Several of the examples in this paper and the case study in Section 6 are based on the *TPC BenchmarkTM C* (TPC-C), which is the standard benchmark for online transaction processing (OLTP). It is a mixture of read-only and update-intensive transactions that simulate the activities found in complex OLTP [10]. Although the TPC-C benchmark application was designed for DBMS performance testing, we chose this application for our case study, in order to exercise our tool set on a real application with a complex schema. The TPC-C schema has multiple composite primary and composite foreign key constraints, some tables have both a composite primary key and one or more composite foreign keys, and some attributes are involved in both a composite primary key and a composite foreign key.

The TPC-C application models a wholesale supplier with a number of geographically distributed warehouses, each serving several sales districts. There are 9 tables (warehouse, district, customer, history, new_order, c_orders, order_line, item, and stock) and 5 transactions (new-orders, payment, order-status, delivery, and stock-level). Figure 1 shows one part of the TPC-C database schema. To save space, attributes that are not relevant to our examples are elided. The warehouse table has information about each warehouse: an ID number, address information, and some accounting information. The primary key constraint says that no two warehouses can have the same ID. The district table has information about the districts and their warehouses: a district ID (*d_id*) and a warehouse ID (*d_w_id*), address and accounting information for the district. The foreign key constraint says that each warehouse ID in the district table must appear as an ID in the warehouse table. The primary key constraint states that no two rows can have the same (district ID, warehouse ID) pair. A key constraint like this, involving more than one attribute, is called a *composite constraint*. The stock and orderline tables keep track of quantities of items in stock and on order, respectively.

3. AGENDA's Database Consistency Check

This section describes AGENDA's technique for checking that a transaction leaves the DB in a consistent state. Complex constraints that are not automatically enforced by

the DBMS are transformed into simpler constraints that are automatically enforced. We first describe the mechanisms the DBMS provides, then present AGENDA's state checking technique, and illustrate it with an example.

3.1. DBMS Integrity Constraint checking

SQL allows the definition of CHECK constraints specifying conditions under which modification to a table is permitted. If a table has a constraint CHECK *condition-expression*, the DBMS will check that the condition-expression is satisfied before inserting, deleting, or updating the table. Condition-expression may be simple constraints, such as permissible range of a data value or may complex SQL expressions. Current commercial DBMSs basically support the entry level of standard SQL92/SQL99, allowing the specification of default values, not NULL constraints, primary and foreign keys, and check constraints involving attributes in the same row [11].

An *assertion* is a Boolean-valued SQL expression that must be true whenever a transaction commits. Assertions can define very general integrity constraints, including constraints involving more than one table and/or with SQL aggregation function. In principle, one could use assertion while testing a transaction to check whether it results in states satisfying all of the semantic constraints (assuming they can be expressed in SQL). Unfortunately, although the concept of an ASSERTION has been introduced in SQL92/SQL99, none of the commercial database management systems currently support this feature fully.

In order to check a constraint that is not enforced by the DBMS, AGENDA creates a temporary table to store the relevant data and converts the assertion into a check constraint at attribute/row level on the temporary table. In particular, constraints involving aggregation functions (SUM, COUNT, etc) and constraints involving multiple tables must be transformed into simpler constraints on auxiliary tables.

3.2 AGENDA's state checking technique

When a transaction commits, it is necessary to check whether the new state satisfies all the global consistency constraints. AGENDA prompts testers to enter global consistency constraints in the form of a precondition and postcondition, each of which can be any logical SQL expression. For example, if the constraint is based on multiple tables, the precondition could specify the join condition on tables. Figure 2 lists one of the twelve global consistency constraints defined in TPC-C Benchmark.

The general procedure to check database state consistency is roughly as follows. When necessary, temporary tables are created to deal with joining relevant attributes from different tables and to replace calls to aggregation

```

CREATE TABLE warehouse ( w_id INT PRIMARY KEY, w_name CHAR(10),
w_street_1 CHAR(20), w_street_2 CHAR(20), w_city CHAR(20), w_state CHAR(2),
w_zip CHAR(9), w_tax MONEY, w_ytd MONEY );

CREATE TABLE district ( d_id INT, d_w_id INT, d_name CHAR(10),
d_street_1 CHAR(20), d_street_2 CHAR(20), d_city CHAR(20), d_state CHAR(2),
d_zip CHAR(9), d_tax MONEY, d_ytd MONEY, d_next_o_id INT,
PRIMARY KEY( d_id, d_w_id), FOREIGN KEY (d_w_id) REFERENCES warehouse(w_id) );

CREATE TABLE stock ( s_i_id INT, s_quantity INT,... );

CREATE TABLE orderLine ( ol_i_id INT, ol_quantity INT,... );

```

Figure 1. partial TPC-C database schema

functions by single attributes representing the aggregate returned. (E.g. a reference to SUM(X) in a constraint gives rise to a column X.SUM in the temporary table, where the sum is stored.) The constraint to be checked is translated into a constraint on a temporary table. SELECT and INSERT statements are generated and executed in order to populate the temporary tables with values corresponding to the original tables. We'd like to check the (translated) constraint on the temporary table, but not all DBMSs allow constraints to be added and dropped dynamically. Instead, the constraint is added to another (initially empty) temporary table and the contents are copied into it, with the constraint checked automatically after each insertion. Constraint violations are reported to AGENDA, indicating that the transaction violated a global state constraint.

Here are the details of the procedure: For each semantic constraint *ic* defined over the global database state:

1. for each table *t* which has aggregation functions over some attributes involved in *ic*

AGENDA creates a temporary table *t'*. The attributes of *t'* consist of attributes involved in *ic*, along with an attribute for each aggregation function. AGENDA generates a SELECT query to retrieve the relevant attributes and aggregates from *t*, then submits the SELECT query to the application DB and saves the result in a cursor. AGENDA generates an INSERT query to store the contents of the cursor into table *t'*.

2. **case A.** *ic* is based on one table *t* and no aggregation function is involved in *ic* :

AGENDA creates a temporary table *t''*. The attributes of *t''* consist of attributes involved in *ic*. The postcondition of *ic* is appended to *t''* as a check constraint. AGENDA generates a SELECT query to retrieve the relevant attributes from *t*;

case B. *ic* is based on one table *t* and there are aggregation functions involved in *ic* :

AGENDA creates a temporary table *t''*. The attributes of *t''* consist of the attributes of *t'*, defined in step 1. The postcondition of *ic* is appended to *t''* as a check constraint; AGENDA generates a SELECT query to retrieve the relevant attributes from *t'*. Note that references to the aggregation function on *t* are replaced by references to the *t'* attribute representing the aggregate.

case C. *ic* is based on multiple tables *t1, ..., tk* and no aggregation function is involved in *ic*:

AGENDA creates a temporary table *t''*. The attributes of *t''* consist of attributes involved in *ic*, and the postcondition of *ic* is appended to *t''* as a check constraint. AGENDA generates a SELECT query to retrieve the relevant attributes from *t1, ..., tk*, where the join condition over *t1, ..., tk* is the precondition of *ic* ;

case D. *ic* is based on multiple tables *t1, ..., tk* and there are aggregation functions involved in *ic*:

AGENDA creates a temporary table *t''*. The attributes of *t''* consist of attributes involved in *ic*, and the postcondition of *ic* is appended to *t''* as a check constraint. AGENDA generates a SELECT query to retrieve the relevant attributes and aggregation functions from *w1, ..., wk*, where *wi* is either *ti* (when there is no aggregation function on *ti*) or *ti'* (when there exists an aggregation function on *ti*), with the precondition of *ic* as the join condition over *w1, ..., wk* .

3. AGENDA submits the SELECT query and saves the result in a cursor, generates an INSERT query to try to store the content of the cursor into table *t''*, and tries to execute the INSERT query. If the insertion fails due to violation of the check constraint, AGENDA captures the exception, and reports the error and the violated constraint. Finally AGENDA removes the temporary tables.

If the DBMS has implemented the feature of dynamically adding/dropping constraints on tables, step 2 could be

simplified. AGENDA could just append the constraint to `t'` directly instead of creating a temporary table `t''`, appending the constraint to `t''` and then trying to copy `t'` to `t''`:

```
ALTER TABLE t' ADD CONSTRAINT
myconstraint AS ic
```

3.3. Example

The constraint in Figure 2 means the year to date revenue for one WAREHOUSE must be equal to the sum of year to date revenue of all its districts. In principle this constraint could be expressed as an assertion:

```
CREATE ASSERTION my_ic CHECK (
(SELECT w_ytd FROM warehouse) = (SELECT
SUM (d_ytd) from district WHERE d_w_id =
warehouse.w_id ) );
```

Unfortunately, assertion is not implemented in today's DBMSs. Below is an example of applying the algorithm in Section 3.2 to the constraint. Figure 3 lists the temporary tables and queries generated.

1. AGENDA generates the schema for table `district_tmp` and the first SELECT query. The temporary table `district_tmp` is used to aggregate the ytd revenues from all the districts corresponding to each warehouse. Each row has a warehouse ID `d_w_id` and the sum of the YTD revenues for all of the corresponding districts, `d_w_ytd_sum`. AGENDA uses a cursor to select the relevant data from the `district` table (using the first SELECT statement in the Figure) and insert it into the `district_tmp` table. The resulting table has one row for each warehouse, storing the warehouse ID and the sum of the ytd revenue for all the districts associated with the warehouse.
2. The constraint involves multiple tables and aggregates (case D). AGENDA generates the schema for table `district_warehouse_tmp`, based on `w1` (`district_tmp`) and `w2` (`warehouse`), with the postcondition "`d_ytd_sum = w_ytd`" appended as a check constraint. Each row of `district_warehouse_tmp` will store a warehouse ID and the sum of revenues for its districts (from table `district_tmp`) and a warehouse ID and the revenue for the warehouse (to be obtained from table `warehouse`). AGENDA generates the second SELECT query whose join condition is the precondition condition "`d_w_id = w_id`". Executing this select query will produce the data to fill table `district_warehouse_tmp`. The INSERT query is generated to store the results from the select into `district_warehouse_tmp`.
3. AGENDA executes the second SELECT query and stores the result in a cursor; AGENDA repeats retriev-

ing a row from the cursor, appending to the INSERT query and executing it. If the constraint is satisfied, table `district_warehouse_tmp` will be successfully filled with (redundant) data for each warehouse. If the constraint is violated for some warehouse (i.e., the ytd entry in the warehouse table is not the sum of ytds of its districts), then the attempt to insert the corresponding row will violate the CHECK constraint, so the insertion will fail and AGENDA will report the problem.

4. AGENDA's Transaction Consistency Check

In addition to resulting in a consistent state, a transaction must transform the DB state in the correct manner. In other words, it must satisfy a *transition constraint* involving two consecutive database states (the states before and after transaction execution).

In earlier work [3] we described a technique for checking a relatively simple transition constraint that involves a single table. AGENDA logs values of relevant attributes before and after execution of the query under test, then checks constraints on the log tables to check whether the query modified the database state appropriately (according to postcondition supplied by the tester). Below is a summary of the procedure to log the state changes. The details of the technique are given in [3].

- AGENDA modifies the schema so that for each table, there is an additional log table that records all modifications made to the table when the application program is executed;
- An SQL *trigger*, also known as an event-condition-action rule, performs some action in response to an event, such as modification of a table, provided some condition is satisfied. AGENDA generates triggers that put entries into the appropriate log table in response to each insert, modify, or delete operation performed on the base tables by the application; These triggers are added to the tables so the DBMS will automatically fill the log tables.

The remainder of this section describes how those techniques were extended to handle more complex constraints (e.g., multiple table constraints with aggregation functions) and how the transition checking is integrated with the global state checking.

4.1 Checking Transitions

The database as a whole has a large state and a transaction usually makes a small incremental change to the state. After the transaction commits, if the log tables are not

Entries in the WAREHOUSE and DISTRICT tables must satisfy the relationship:

$W_YTD = \text{SUM}(D_YTD)$ for each warehouse defined by $(W_ID = D_W_ID)$.

Figure 2. Sample global consistency constraint

```
CREATE TABLE district_tmp (d_w_id INT, d_w_ytd_sum MONEY);

SELECT d_w_id, SUM(d_ytd) FROM district GROUP BY d_w_id;

CREATE TABLE district_warehouse_tmp (d_w_id INT, d_w_ytd_sum MONEY, w_id INT,
w_ytd MONEY, CHECK (d_ytd_sum=w_ytd ) );

SELECT d_w_id, d_ytd_sum, w_id, w_ytd FROM district_tmp, warehouse WHERE d_w_id=w_id;

INSERT INTO district_warehouse_tmp (d_w_id, d_w_ytd_sum, w_id, w_ytd) values (...);
```

Figure 3. Temporary tables and queries for checking state constraint

empty, not only do we need to check if all the global consistency properties hold, but also we need to verify if the new state satisfies the requirement of the transaction's specifications by checking how the state changed. AGENDA checks the constraint as follows:

- AGENDA generates a temporary table based on the tables/attributes and old/new values of attributes, with the post-condition as a check constraint on the temporary table, then generates a SELECT query based on the precondition of the constraint, and saves the result in a cursor;
- AGENDA generates an INSERT query to try to store the content of the cursor into the temporary table. If the insertion fails due to violation of the check constraint, AGENDA captures the exception, and reports the error and the violated constraint. Finally AGENDA removes the temporary tables.

For example, one constraint in the NEW_ORDER transaction for TPC-C benchmark is

the sum of quantity for one item in tables STOCK and ORDER_LINE must remain unchanged.

Figure 4 lists the temporary table and query generated. The order_line_stock_tmp table is generated to store relevant attributes from the log tables created for order_line and stock. The check constraint on that table comes from the postcondition of the transition constraint. Note that it involves values from before and after the transaction (marked old and new, respectively) and attributes from different tables. The SELECT statement selects the relevant attributes from the log tables. AGENDA

stores the result of the SELECT into a cursor, then fills the order_line_stock_tmp table, with the constraint being automatically checked for each row. If the sum of quantities in the stock and orderline table has changed for some item, the check constraint will be violated when the corresponding row is inserted and AGENDA will report the error.

4.2 Two levels of checking complement each other

An application may contain many global consistency constraints. At first all these constraints are validated for the initial DB. After a transaction commits, AGENDA validates whether the transition satisfies the requirement for the transaction specification. AGENDA does not check all the global constraints again. Rather, it checks whether the log tables for all the application tables associated with each global constraint are empty. If at least one of them is not empty, this means the new state for these tables is different than the old one, so the global consistency constraint needs to be checked again for the new state.

These two levels of checks complement each other. For example, if the transaction which should update one table fails, nothing changes in the DB state. So all the global consistency constraints hold. The transaction specification should state that this table should be updated, and the transition check reports that this table did not change. Then the error is detected. On the other hand, if the transaction updates a table besides those which are allowed in its specification, then the transition check can not find any problem. Only by checking the global consistency properties, AGENDA might find the problem. In summary, state checking and transition checking complement each other, and

```

CREATE TABLE order_line_stock_tmp(ol_iid_old INT, ol_quantity_old INT, ol_iid_new
INT, ol_quantity_new INT, s_iid_old INT, s_quantity_old INT, s_iid_new INT,
s_quantity_new INT, check ( s_quantity_old + ol_quantity_old
= s_quantity_new + ol_quantity_new));

SELECT ol_iid_old, ol_quantity_old, ol_iid_new, ol_quantity_new, s_iid_old,
s_quantity_old, s_iid_new, s_quantity_new FROM order_line_log, stock_log WHERE
s_iid_old = ol_iid_old;

```

Figure 4. Temporary table and query for checking transition constraint

work together to check consistency efficiently.

5. DB State and Test Case Generation

5.1. Update of the initial DB

In the previous work, AGENDA generates an initial DB, which satisfies all the not NULL, uniqueness, and referential integrity constraints defined in the schema, but it might not satisfy the semantic constraints defined over the global state. Here AGENDA is extended to deal with global consistency constraints. Based on information about the state constraint, AGENDA updates the initial DB to satisfy the constraint. The general procedure is as follows:

- If there is an aggregation function, AGENDA generates temporary tables based on the tables/attributes in the above steps, then generates queries to populate the temporary tables.
- AGENDA generates an UPDATE query to update the initial DB to satisfy the consistency property.

For the state constraint defined in Section 3.3, AGENDA runs the above procedure to update table WAREHOUSE. Figure 5 lists the temporary table and queries generated.

In order to facilitate interaction with the tester, AGENDA can capture the constraint via GUI as follows :

A. The tester specifies the tables (DISTRICT, WAREHOUSE) involved in the constraint.

B. The tester further chooses the involved attributes (W_ID, W_YTD, D_W_ID, D_YTD) in the chosen tables. Optionally, the tester specifies if there is a "group by" operation based on the attribute (D_W_ID) or if there is an aggregation function for each attribute (aggregation function SUM for D_YTD). The available aggregation functions include SUM, AVG, COUNT, MAX, and MIN.

C. The tester provides the preconditions ($D_W_ID = W_ID$) and post-conditions ($W_YTD = D_YTD_SUM$) based on the specified tables/attributes .

Alternatively, the constraint can be supplied via a batch file.

In this approach, AGENDA needs to know the semantics about the table/attribute to be updated and how the update should be. For some constraints, these semantics are not expressed clearly; for other constraints, it is necessary to insert/delete some tuples in some tables in order to satisfy the constraint. Currently AGENDA can not handle either of these situations. The current technique is sufficiently powerful to handle most constraints in TPC-C, which are fairly complex.

5.2 Input Generation

The general procedure for the Input Generator is as follows:

1. Get heuristics for input generation.
2. While not done generating test cases
 - (a) For each input parameter that needs to be instantiated
 - i. Get relevant information (table, attribute) associated with this parameter from the Agenda DB.
 - ii. Choose a data group and value according to the heuristics chosen.
 - iii. Update bookkeeping information regarding progress achieved toward satisfying each heuristic.
 - iv. Set done = true if all heuristics are satisfied, or if we cannot proceed (generated the maximum number of test cases).
 - (b) Map the test case generated by step (a) to a template, in the Agenda DB.

If the Input Generator chooses values arbitrarily, then the data generated may cause at least one of the application's queries to return an empty table as the result. For example, suppose an application query has the following in its WHERE clause: $emp.empno = :in_empno$. If the Input Generator knowingly chooses a value for in_empno for which

```

CREATE TABLE district_tmp (d_w_id INT, d_w_ytd_sum MONEY);

SELECT d_w_id, SUM(d_ytd) FROM district;

UPDATE warehouse SET w_ytd =(SELECT d_ytd_sum FROM district_tmp WHERE d_w_id=w_id);

```

Figure 5. Temporary tables and queries for updating the initial db

the query will return an empty table, this is fine for testing the application’s robustness (i.e. making sure that the application does not crash on a subsequent statement which accesses the empty table), but we also want to test the application on the kinds of data that would normally be expected. In this example, this means choosing a value for `:in_empno` which appears in the database for attribute `empno` of table `emp`. This would increase the likelihood that the transaction under test commits/completes, thus exploring the application’s handling of this scenario. A test case which is constructed with an effort to increase the likelihood of the transaction committing is classified as *type A*. Ideally, a *type A* test case satisfies all the `WHERE` clauses involving input parameters (if possible) so that if the transaction under test is correct, it will commit; if it does not commit, then there is an error in the transaction; if it commits, it may or may not be correct. A test case which is constructed with an effort to decrease the likelihood of the transaction committing is classified as *type B*.

Generating *type A* test cases becomes harder when we consider composite constraints. For example, suppose an application query has the following in its `WHERE` clause: `a = :input1` and `b = :input2` and that attributes `a` and `b` are involved in a composite primary/unique/foreign key. For a *type A* test case, the Input Generator chooses values for `input1` and `input2` such that these values appear in the same tuple of the table on which the composite key on their associated attributes exists.

Generating *type A* test cases is further challenged by dependent parameters. For example, suppose an application query has the following in its `WHERE` clause: `c = :input3` and `d = :input4` and that `c` has a primary key but `d` has no key constraint; the value chosen for `input4` still depends on the value chosen for `input3`, in a *type A* test case. Furthermore, a parameter’s value may depend on more than one attribute, if those attributes are involved in a composite key. TPC-C benchmark has many instances of input parameters associated with attributes involved in composite primary and/or composite foreign keys, as well as input parameters whose values are dependent on values selected for other input parameters (associated with attributes involved in primary key or composite constraints). AGENDA’s handling of key constraints, composite constraints, and dependent parameters, in generating *type A* test cases, is discussed in detail and illus-

trated with examples in [1].

6. Case study based on TPC-C benchmark

The TPC-C benchmark is implemented in C with embedded SQL. A graduate student (not one of the authors) created buggy versions by seeding a diverse set of SQL errors that he considered common and realistic. A description of each seeded error is provided in Figure 6. For each of 12 buggy versions (labelled E1, E2, E3, ..., E12 respectively) of the TPC-C application, AGENDA generated a database state (45 rows for 9 tables; all data groups represented) and generated test cases (according to the *type A* and *all groups* heuristics).

Each buggy implementation contained a single error, among E1 through E12, and test data was generated for each implementation separately, since in reality, we do not have the correct implementation. As a sanity check, we also generated test cases for the correct implementation to check that the tool was indeed producing *type A* test cases. There were no false positives, in that all of the test cases generated for the correct implementation, when executed on the correct implementation, resulted in the transaction committing (in accordance with the definition and purpose of *type A*).

When a *type A* test case is executed by an application under test (buggy implementation) and the transaction does not commit, this indicates an error in the implementation. In Figure 6, an entry of “Y” in column “RESULT” means that AGENDA exposed the error. An entry of “N” in column “RESULT” means that the error was not exposed. Columns labeled “PAR”, “COM”, “IC1”, and “IC2” give more detail about the kind of constraint violated. An entry of ‘Y’ in the column labeled “PAR” means that the error is a syntax error and can be detected by the SQL parser. Versions E5, E8 and E19 have syntax errors. All other nine versions have semantic errors. First, each version was tested without state checking or transition checking enabled. An entry of “Y” in the column labeled “COM” indicates that the transaction did not commit in this situation. An entry of “N” in column “COM” means that the buggy transaction committed, and thus the error was not exposed yet. As detailed below, error detection resulting from failure to commit could happen when a constraint that’s enforced by the DBMS is violated or when the results of one query lead to incorrect inputs to

VER	Description	PAR	COM	IC1	IC2	RESULT
E1	missing condition in WHERE clause	N	Y			Y
E2	missing an INSERT statement	N	N	Y		Y
E3	same SELECT query appears twice in a row	N	N	N	N	N
E4	missing last column of INSERT statement	N	N		Y	Y
E5	too few arguments in SELECT clause	Y	Y			Y
E6	SELECT on a value outside of expected domain	N	Y			Y
E7	result of query stored in different variable	N	N		Y	Y
E8	too few host variable in INTO clause	Y	Y			Y
E9	database not connected	N	Y			Y
E10	extra condition in WHERE clause	N	N	N	N	N
E11	non-int attributes selected INTO int variable	Y	Y			Y
E12	updating host variable incorrectly	N	N		Y	Y

Figure 6. Test result based on TPC-C

a subsequent query. The columns labeled “IC1” and “IC2” indicate errors that were not detected in the first phase, but were detected by checking global state constraints or transition constraints, respectively.

At first glance, it might appear that for some SQL faults that occur on all paths, as in version E11, any test case would expose the fault. However, if the transaction under test has n queries and there is an fault in the k -th query, then a test case, in order to expose this fault, must pass through the first $k-1$ queries. Moreover, if the fault occurs on some (not all) path(s), as in E5, E6, and E8, then in order to expose the fault, some input test case needs to be generated that forces execution of the path containing the fault. Errors in versions E5, E6 and E8 were exposed by at least one test case due to appropriate partitioning of data groups, as well as the tool’s ability to represent all groups and generate *type A* test cases. Version E1 involved a missing condition in a WHERE clause; the missing condition contained an attribute involved in a composite constraint, thus causing a subsequent composite constraint violation, given AGENDA’s test cases. Version E9 has an anomalous fault, since the application, executing with this fault (database not connected), cannot commit regardless of the test case. All other faults were seeded by modifying SQL queries in the TPC-C application. The faults in versions E1, E5, E6, E8, E9, and E11 were exposed because the transactions containing these faults failed to commit on at least one of the *type A* test cases generated, as discussed above. Further details are provided in [1]. The *type A* test cases commit on the oracle version and versions E2, E3, E4, E7, E10 and E12. Version E3 is not incorrect, but inefficient; the same SELECT query is executed twice in succession. Though semantic faults in versions E1 and E11 were exposed by *type A* test cases (due to constraint violations in subsequent queries), in general, the parsing and generation tools (Agenda Parser, State Generator and Input Generator)

cannot by themselves expose semantic faults; however, by incorporating preconditions and postconditions in conjunction with the test data generated, the checking tools (State Validator and Output Validator) can expose semantic faults when a property is violated, as follows. In version E2 and E12 the fault is exposed by the second global consistency constraint. The fault in version E4 is exposed by a simple transaction constraint: “the value for attribute `ol_dist_info` in table `ORDER_LINE` can not be null”. The fault in version E7 is detected by a transaction constraint: “the balance must be changed in the `CUSTOMER` table after a new order is made by the customer”. The bug in version E10 is not captured. This bug is an extra condition in the WHERE clause of a SELECT query in transaction `STOCK-LEVEL`. To capture this fault, both the database state and inputs need to be generated to satisfy the desired requirements, and with the suitable precondition/postcondition. There is no change in the database state for this transaction. The last column in Figure 6 indicates that AGENDA’s tools exposed 10 of the 12 seeded faults.

7. Related Work

Reference [7] states that database integrity has two complementary components: validity, which guarantees that all false information is excluded from the database, and completeness, which guarantees that all true information is included in the database. It describes a uniform model of integrity for relational database, that considers both validity and completeness.

Transactions that modify the database must insure that business rules are never violated. Such transactions are called “safe”. Pdiff is a tool targeted to the software development process for the ATT 5ESS switch. It takes a constraint and a possibly unsafe transaction as input, and

produces a safe transaction as output. A first-order specification language PRL is used to generate update constraints in Pdiff [5].

A transaction logic for the specification of the dynamic behavior of databases is introduced in [8]. The evolution of databases is characterized by both the dynamic integrity constraints which describe the properties of state transactions and the transactions whose execution lead to state transitions. The formalism is based on a variant of first-order situational logic in which the states of computations are explicit objects. Integrity constraints and transactions are uniformly specifiable as expressions.

Tim Sheard and David Stemple built a system to prove at compile-time that transactions cannot, if run atomically, disobey integrity constraints. The system performs such automatic verification for a robust set of constraints and transaction classes. It accepts database schemas written in a more or less transactional style and accepts programs in a high-level programming language [9].

All the above approaches use some specific formal specification languages. In contrast to them, in the AGENDA system, the tester specifies the precondition/post-condition with simple operations via GUI; AGENDA automatically translates the condition into standard SQL constraints and the DBMS automatically checks these constraints.

8. Conclusions and future work

In response to a lack of existing approaches specifically designed for testing database applications, we have proposed a framework for that purpose, and have designed and implemented a tool set to partially automate the process. Transaction is the basic unit of a database application. It is very important to insure that a transaction satisfies the ACID properties. Databases are typically required to satisfy a collection of integrity constraints—conditions that specify the valid states of the database. Database transactions that update the database must preserve these constraints and transform the DB according to the transaction specification. In this paper, we extend AGENDA to test database transaction consistency. The internal constraint and business rules are captured through simple GUI operations. The initial database state is updated to reflect these business rules and internal constraints. After the transaction commits, two levels of checking are used to check database consistency. The transition check validates that the transition of transaction satisfies its specification. The state check validates that the overall global consistency properties hold for the new database states. The transition check and global state check complement each other and work together to check the application consistency efficiently. Complexity constraints are checked via creating two levels of temporary table and converting general SQL assertions into check constraints on the

temporary table. We believe the two level check techniques will scale well, as only the necessary relevant constraints are checked when the changes in the log table might affect these constraints after one transaction commits. In a case study based on TPC-C benchmark, our techniques are quite effective.

Currently, the database consistency properties are specified by the tester via GUI; we are exploring techniques to integrate some practical specification techniques such as UML or BON into our AGENDA system, and automatically convert the properties in the specification into SQL constraints; then the output checking in the AGENDA system could be fully automated. A Java/JDBC front end is under development. We also plan more extensive empirical evaluation.

References

- [1] D. Chays. *Test Data Generation for Relational Database Applications*. PhD thesis, Computer and Information Science, Polytechnic University, 2003.
- [2] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weyuker. A framework for testing database applications. *Proceedings of the 2000 International Symposium on Software Testing and Analysis*, pages 147–157, Aug. 2000.
- [3] D. Chays, Y. Deng, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weyuker. An agenda to test relational database application. *Journal of Software Testing, Verification and Reliability*, 2004.
- [4] Y. Deng, P. Frankl, and Z. Chen. Testing database transaction concurrency. In *IEEE International Conference on Automated Software Engineering 2003*. IEEE Computer Society Press, Oct. 2003.
- [5] T. Griffin and H. Trickey. Integrity maintenance in a telecommunications switch. In *IEEE Data Engineering Bulletin*, pages 43–46. IEEE Computer Society Press, June 1994.
- [6] P. Lewis, A. Bernstein, and M. Kifer. *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley, 2002.
- [7] A. Motro. Integrity = validity + completeness. *ACM Transactions on Database Systems*, pages 480–502, Dec. 1989.
- [8] X. Qian. A transaction logic for database specification. In *International Conference on Management of Data and Symposium on Principles of Database Systems*, pages 243–250. ACM Press, June 1988.
- [9] T. Sheard and D. Stemple. Automatic verification of database transaction safety. *ACM Transactions on Database Systems*, pages 322–368, Sept. 1989.
- [10] Transaction Processing Performance Council. *TPC-Benchmark C*. 1998.
- [11] C. Turker and M. Gertz. Semantic integrity support in SQL: 1999 and commercial (object-)relational database management systems. *Very Large Data Bases*, Sept. 2001.