Testing of the data access layer and the database itself

Vineta Arnicane and Guntis Arnicans

University of Latvia

TAPOST 2015, 08.10.2015

Prolog

Vineta Arnicane, Guntis Arnicans, Girts Karnitis

DigiBrowser as a tool for testers

is replaced by

Vineta Arnicane and Guntis Arnicans

Testing of the data access layer and the database itself

Database application testing

- «A **database application** is a computer program whose primary purpose is **entering and retrieving information from a computerized database**» (Wikipedia)
- Tons of applications are developed
  - How do developers perform testing?
  - Is adequate attention granted to testing of database itself and interactions with it?

Typical testing approach

- Approach:
  - take an application as a black box software
  - design test cases according to formal or informal specification
  - [prepare database initial state]
  - run test cases
  - verify results (software behavior)
- It is a black box testing!
  - sometimes testers have no access to source code
  - sometimes testers have no programming skills or they are not IT specialists at all

Database testing in Wikipedia

- Term «database testing» is not too popular (the Wikipedia article is created 5 December 2011)
- There are many interpretations what this term means (Wikipedia does not provide a clear definition at all)
- «It is important to test in order to obtain a database system which satisfies the ACID properties (Atomicity, Consistency, Isolation, and Durability) of a database management system»
- «Database testing usually consists of a layered process, including the user interface (UI) layer, the business layer, **the data access layer and the database itself**»

What is database testing? I

Source: http://testsoftwarefaq.blogspot.com/2014/02/database-testing-faq.html

- Database testing involves some in depth knowledge of the given application and requires more defined plan of approach to test the data
- Key issues include:
  1) Data integrity
  2) Data validity
  3) Data manipulation and updates
- Tester must be **aware of the database design concepts and implementation rules**

What is database testing? II

Source: http://testsoftwarefaq.blogspot.com/2014/02/database-testing-faq.html

- Database testing is all about testing:
  - ✓ joins,
  - ✓ views,
  - ✓ imports and exports,
  - ✓ testing the procedures,
  - ✓ checking locks,
  - ✓ indexing,
  - ✓ etc.
- It’s not about testing the data in the database (!)
- Usually database testing is performed by DBA (!)

What is database testing? III

Source: http://testsoftwarefaq.blogspot.com/2014/02/database-testing-faq.html

- Database testing basically includes the following:
  1) Data validity testing
  2) Data integrity testing
  3) Performance related to data base
  4) Testing of procedure, triggers and functions
- For doing data validity testing **you should be good in SQL queries**
- For data integrity testing you **should know about referential integrity and different constraints**
- For performance related things you **should have idea about the table structure and design**
- For testing procedures, triggers and functions you should be able to understand the same

Using of SQL statements in database testing

Source: http://testsoftwarefaq.blogspot.com/2014/02/database-testing-faq.html

- **The most important statement for database testing is the SELECT statement**, which returns data rows from one or multiple tables that satisfy a given set of criteria
- **You may need to use other DML** (Data Manipulation Language) statements like INSERT, UPDATE and DELETE to manage your test data
- **You may also need to use DDL** (Data Definition Language) statements like CREATE TABLE, ALTER TABLE, and DROP TABLE to manage your test tables
- **You may also need to some other commands** to view table structures, column definitions, indexes, constraints and stored procedures

Types of database testing

- **Structural testing**
  - Schema
  - Database elements - tables, columns
  - Default values for a column
  - Data invariants for a single column
  - Data invariants involving several columns
  - Referential integrity rules
  - Stored procedures/functions
  - Triggers
  - Views testing
  - Constraints
  - Database server validations
  - Existing data quality

- **Functional database testing**
  - Checking data integrity and consistency
  - Login and user security
  - Incoming data values
  - Outgoing data values (from queries, stored functions, views ...)

- **Non-functional testing**
  - Load testing
  - Stress testing

Database functional testing tasks

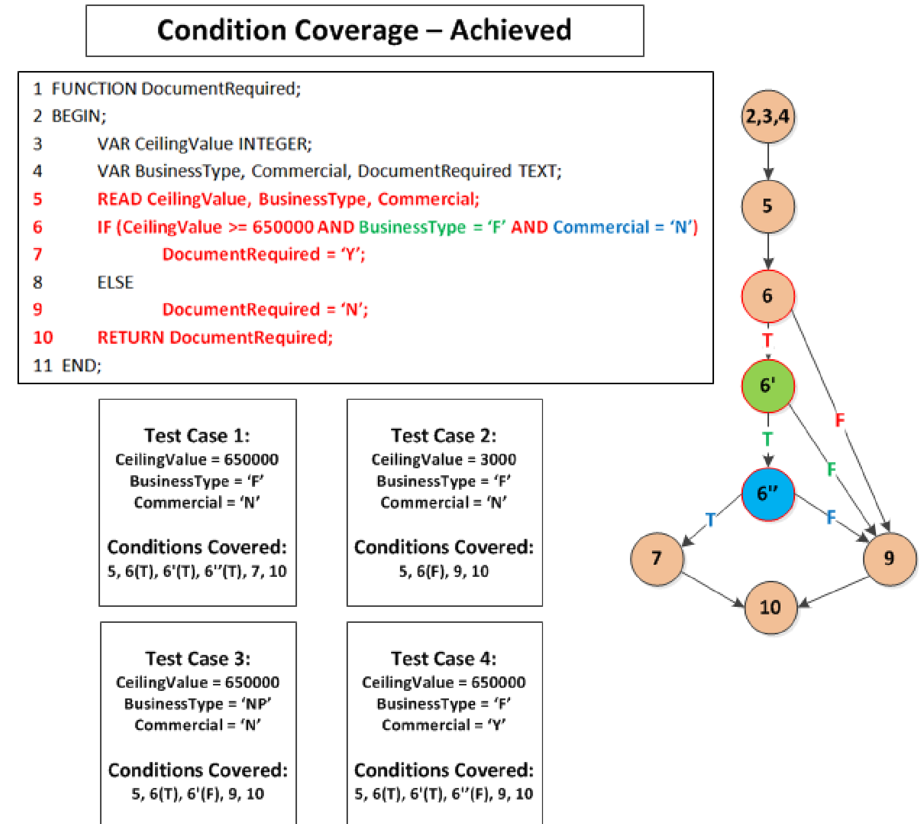
- **Database initialization** - put database into a known state before running tests to make sure that the tests will be executed correctly
- **Functional testing** - check that the application operates with the database correctly
- **Data verification** - check the structure and the actual content of a database

Necessity for white box testing

- Black box testing **cannot reveal all defects** in software
- White box testing **allows to find specific problems** and to evaluate a testing quality by exploiting various testing **coverage metrics**
- But it **requires good IT skills** and understanding what we can do and what we cannot, and how many resources it takes

Software testing coverage (ISTQB)

- statement coverage
- branch coverage
- condition coverage
  - decision condition coverage
  - multiple condition coverage
- decision coverage
  - modified condition / condition coverage (MC/DC)
- LCSAJ coverage
- path coverage
- N-swich coverage
- boundary value coverage
- equivalence partition coverage
- data flow coverage



Source: http://www.seguetech.com/blog/2014/06/13/how-much-test-coverage-enough-testing-strategy

SQL statement itself is like a program

select e.company, c.city, **sum**(e.salary)/**count**(e.id) **as** avg_salary

from employees e, cities c

where e.city_id = c.id **and** e.age > 30 **and** e.company **like** «%Microsoft%»

group by company, city

order by avg_salary **desc**

having avg_salary < 2000

SQL statement itself is like a program

```
select e.company, c.city, sum(e.salary)/count(e.id) as avg_salary
        from employees e, cities c
        where e.city_id = c.id and e.age > 30 and  e.company like «%Microsoft%»
        group by company, city
        order by avg_salary desc
        having avg_salary < 2000
```

- Is it enough to «cover» this statement only with one test case?
- No, of course!

- Wait! What is a test case?
- This statement maybe depend on input data or maybe do not. It also depends on a database state while its executing

SQL statement itself is like a program

```
select m.company, c.id, c.city, sum(e.salary)/count(e.id) as avg_salary
        from employees e, cities c, companies m
        where e.city_id = c.id and e.company_id = m.id and
                e.age > 30 and  m.company like «%Microsoft%»
        group by company, city
        order by avg_salary desc
        having avg_salary < 2000
```

- Mistakes using table identifiers (letters):
  - ✓ It is easy use wrong letter (**c**ities **c**, **c**ompanies **m**)
  - ✓ wrong table for record counting (**c**.id or **m**.id instead of **e**.id in **count**(e.id))

SQL statement itself is like a program

select m.company, c.id, c.city, **sum**(e.salary)/**count**(e.id) **as** avg_salary
from employees e, cities c, companies m
where e.city_id = c.id **and** e.company_id = m.id **and**
e.age > 30 **and** m.company **like** «%Microsoft%»
group by company, city
order by avg_salary **desc**
having avg_salary < 2000

- Wrong conditions:
  - ✓ e.age >= 30, e.age < 30, e.age > 20, e.age > 40, e.age > 60
  - ✓ wrong operator **and**, **or**, **not** (e.g. **not like**)
  - ✓ subcondition is not included into parenthesis
  - ✓ Upper/lower letters (**M**icrosoft, **M**icrosoft)
  - ✓ Whole word (**Microsoft**ware, Super**Microsoft**)

SQL statement itself is like a program

```
select m.company, c.id, c.city, sum(e.salary)/count(e.id) as avg_salary
    from employees e, cities c, companies m
    where e.city_id = c.id and e.company_id = m.id and
        e.age > 30 and  m.company like «%Microsoft%»
    group by company, city
    order by avg_salary desc
    having avg_salary < 2000
```

- Additional conditions (having):
  - ✓ avg_salary <2000
  - ✓ avg_salary <**=**2000
  - ✓ avg_salary **>**2000
  - ✓ avg_salary <**3**000

SQL statement itself is like a program

select m.company, c.id, c.city, **sum**(e.salary)/**count**(e.id) **as** avg_salary
from employees e, cities c, companies m
where e.city_id = c.id **and** e.company_id = m.id **and**
e.age > 30 **and** m.company **like** «%Microsoft%»
group by company, city
order by avg_salary **desc**
having avg_salary < 2000

- Aggregation functions:
  - ✓ Is it possible that **count**(e.id) returns 0?
  - ✓ Maybe e.salary has different currencies in different records?

SQL statement itself is like a program

```
select m.company, c.id, c.city, sum(e.salary)/count(e.id) as avg_salary
        from employees e, cities c, companies m
        where e.city_id = c.id and e.company_id = m.id and
                e.age > 30 and  m.company like «%Microsoft%»
        group by company, city
        order by avg_salary desc
        having avg_salary < 2000
```

- Join problems in where clause:
  - ✓ NULL values in fields (e.city_id, e.company_id)
  - ✓ inner, left or right join

  - ✓ missed records (database integrity is broken)
  - ✓ excessive records (duplicates) cause wrong result

SQL statement itself is like a program

```
select m.company, c.id, c.city, sum(e.salary)/count(e.id) as avg_salary
        from employees e, cities c, companies m
        where e.city_id = c.id and e.company_id = m.id and
                e.age > 30 and  m.company like «%Microsoft%»
        group by company, city
        order by avg_salary desc
        having avg_salary < 2000
```

- SQL SELECT statement has many commands/functions/options
- Different DBMS use different SQL standards with «specific features»
- For adequate SQL statement testing we need:
  - ✓ various database states (deciding what data we need; generating needed records, initializing database for each «test case»)
  - ✓ oracle for evaluating execution result

Coverage criteria for testing SQL queries

- Good criteria helps to create needed database states and optimize number of states
- Only a few groups of researchers worked on this issue
- Criteria are developed for the most popular cases
- Most of testers do not know about criteria
- There are no well-described cases of usage in industry
- Automation or tool support is weak

Principles of criteria (JOIN)

from employees e, cities c, companies m
where e.city_id = c.id **and** e.company_id = m.id

- We need database states for «e.city_id = c.id» where
  - ✓ no records in both tables
  - ✓ no records in one table
  - ✓ city_id has NULL and not NULL values
  - ✓ exactly one matching
  - ✓ many matchings (usual number, large number)
  - ✓ no matchings
  - ✓ combinations of previous requirements
- Condition coverage for e.city_id = c.id **and** e.company_id = m.id
  FF, FT, TF, TT
- Combinations of both mentioned groups

Principles of criteria (condition)

e.age > 30 **and** m.company **like** «%Microsoft%»

- We need database states for «e.age > 30» where
  - ✓ no matchings
  - ✓ exactly one matching
  - ✓ many matchings (usual number, large number)
  - ✓ combinations of previous requirements
- Condition coverage for e.age > 30 and m.company like «%Microsoft%»
  FF, FT, TF, TT
- Combinations of both mentioned groups

Principles of criteria (other)

group by company, city

order by avg_salary **desc**

having avg_salary < 2000

- For each clause we can define similar coverage requirements

Dynamically created SQL statement

```
«select m.company, c.id, c.city, sum(e.salary)/count(e.id) as avg_salary
        from employees e, cities c, companies m
        where e.city_id = c.id and e.company_id = m.id and
                e.age > » + ageVar + « and  m.company like ‘%» + companyVar + «%’
        group by company, city
        order by avg_salary desc
        having avg_salary < » + salaryVar
```

- Symbolic execution can help us to obtain all or significant part of all possible SQL SELECT statements
- If input values goes from database or user, then number of variants can be infinite

Mutation testing

```
select m.company, c.id, c.city, sum(e.salary)/count(e.id) as avg_salary
        from employees e, cities c, companies m
        where e.city_id = c.id and e.company_id = m.id and
                e.age > 30 and  m.company like «%Microsoft%»
        group by company, city
        order by avg_salary desc
        having avg_salary < 2000
```

- Make a mutation of the original statement, e.g. e.age >**=** 30
- Execute all test cases
- If all results are the same as for the original statement, then test suite is not adequate

- Do these steps for all possible mutations

Other SQL statements

- Insert
- Delete
- Update
- Etc.

- SELECT is read only operation
- Most of other statements change database schema or data
- We need to verify changes in the database!

Conclusions

- Database testing problem is hard and huge resource consuming
- Nor researchers, nor practitioners have solutions for adequate database testing
- A lot of theoretical results are unknown for practitioners
- There are not many tools supporting at least part of database testing activities
  - Most of tools are developed in universities and are not maintained

Epilogue

- DigiBrowser is[/was] a tool that can help relational database exploring un data inspecting more easily then other data browsers
- DigiBrowser does not require writing SQL queries and can by used by non-IT specialists

Thanks!