Tutorial Database Testing using SQL

1. INTRODUCTION

This document is to discuss general test specification issues for SQL server back end testing and to provide testers a test design guide that includes test methodology.

Most systems, i.e. Forecast LRS, Delta, KENAI, KBATS and so on, that are developed by ITG have client-server architectures. However, only a few projects have their back end completely tested.

1.1 Why back end testing is so important

A back end is the engine of any client/server system. If the back end malfunctions, it may cause system deadlock, data corruption, data loss and bad performance. Many front ends log on to a single SQL server. A bug in a back end may put serious impact on the whole system. Too many bugs in a back end will cost tremendous resources to find and fix bugs and

delay the system developments.

It is very likely that many tests in a front end only hit a small portion of a back end. Many bugs in a back end cannot be easily discovered without direct testing.

Back end testing has several advantages: The back end is no longer a "black box" to testers. We have full control of test coverage and depth. Many bugs can be effectively found and fixed in the early development stage. Take Forecast LRS as an example; the number of bugs in a back end was more than 30% of total number of bugs in the project. When back end bugs are fixed, the system quality is dramatically increased.

1.2 Differences between back end testing and front end testing

It is not easier to understand and verify a back end than a front end because a front end usually has friendly and intuitive user interfaces.

A back end has its own objects, such as, tables, stored procedures and triggers. Data integrity and protection is critical. Performance and multi-user support are big issues. Slowness in operation can be vital to the project's future.

There are no sufficient tools for back end testing. SQL language is mainly a testing tool. MS Access and MS Excel can be used to verify data but they are not perfect for testing. However, there are a large number of test tools available for front end testing.

To be able to do back end testing, a tester must have strong background in SQL server and SQL language. It is relatively difficult to find testers who understand both SQL server and SQL testing. This causes a shortage of back end testers.

1.3 Back end testing phases

There are several phases in back end testing. The first step is to acquire design specifications for an SQL server. The second step is test specification design. The next step is to implement the tests in this design with SQL code. The test specification design should contain information concerning component testing (individual pieces of the system), regression testing (previously known bugs), integration testing (several pieces of the system put together), and then the entire system (which will include both front and back ends).

Component testing will be done early in the development cycle. Integration and system tests (including interfaces to front ends and nightly processes) are performed after the component tests pass. Regression testing will be performed continuously throughout the project until it is finished. The back end usually does not have an independent beta test, as it only exercised by the front end during the beta test period. The last step is to deliver users a quality product.

1.4 Back end test methodology

Back end test methodology has many things in common with front end testing and API testing. Many test methods can be used for back end testing. Structural testing and functional testing are more effective approaches in back end testing. They are overlapped in some test cases. However, the two methods may discover different bugs. We strongly recommend testers to do both types of testing.
There are many other test methods that can be applied to back end testing. We list a few below. For other test methods, please check other test design references.

Structural testing:

A back end can be broken down into a finite number of testable pieces based on a back end's structure. Tests will verify each and every object in a type of structure.

Functional testing:

A back end can be broken down into a finite number of testable pieces based on application's functionality. The test focus is on functionality of input and output but not on the implementation and structure. Different projects may have different ways to break down.

Boundary testing:

Many columns have boundary conditions. For example, in a column for percentages, the value cannot be less than zero and cannot be greater than 100%. We should find out these types of boundary conditions and test them.

Stress testing:

It involves subjecting a database to heavy loads. For incidence, many users heavily access the same table that has a large number of records. To simulate this situation, we need to start as many machines as possible and run the tests over and over.

2. STRUCTURAL BACK END TESTS

Although not all databases are the same, there are a set of test areas that will be covered in all test specifications.

Based on structure, a SQL database can be divided into three categories: database schema, stored procedures, and triggers. Schema includes database design, tables, table columns, column types, keys, indices, defaults, and rules. Stored procedures are constructed on the top of a SQL database. The front end talks to APIs in DLL. The APIs communicate a SQL database through those stored procedures. Triggers are a kind of stored procedures. They are the "last line of defense" to protect data when data is about to be inserted, updated or deleted.

Figure 1. The structure of SQL back end

2.1 Database schema testing

Test Coverage Criterion: *"EACH AND EVERY ITEM IN SCHEMA MUST BE TESTED AT LEAST ONCE"*

2.1.1 Databases and devices

Verify the following things and find out the differences between specification and actual databases

- Database names
- Data device, log device and dump device
- Enough space allocated for each database
- Database option setting (i.e. trunc. option)

2.1.2 Tables, columns, column types, defaults, and rules

Verify the following things and find out the differences between specification and actual tables

- All table names
- Column names for each table
- Column types for each table (int, tinyint, varchar, char, text, datetime. specially the number of characters for char and varchar)
- Whether a column allows NULL or not
- Default definitions
- Whether a default is bound to correct table columns
- Rule definitions
- Whether a rule is bound to correct table columns
- Whether access privileges are granted to correct groups

2.1.3 Keys and indexes,

Verify the following things and compare them with design specification

- Primary key for each table (every table should have a primary key)
- Foreign keys
- Column data types between a foreign key column and a column in other table
- Indices, clustered or nonclustered; unique or not unique

2.2 Stored procedure tests

Test Coverage Criterion: *"EACH AND EVERY STORED PROCEDURE MUST BE TESTED AT LEAST ONCE"*

2.2.1 Individual procedure tests

Verify the following things and compare them with design specification

- Whether a stored procedure is installed in a database

- Stored procedure name
- Parameter names, parameter types and the number of parameters

Outputs:

- When output is zero (zero row affected)
- When some records are extracted
- Output contains many records
- What a stored procedure is supposed to do
- What a stored procedure is not supposed to do
- Write simple queries to see if a stored procedure populates right data

Parameters:

- Check parameters if they are required.
- Call stored procedures with valid data
- Call procedures with boundary data
- Make each parameter invalid a time and run a procedure

Return values:

- Whether a stored procedure returns values
- When a failure occurs, nonzero must be returned.

Error messages:

- Make stored procedure fail and cause every error message to occur at least once
- Find out any exception that doesn't have a predefined error message

Others:

- Whether a stored procedure grants correct access privilege to a group/user
- See if a stored procedure hits any trigger error, index error, and rule error
- Look into a procedure code and make sure major branches are test covered.

2.2.2 Integration tests of procedures

- Group related stored procedures together. Call them in particular order

- If there are many sequences to call a group of procedures, find out equivalent classes and run tests to cover every class.

- Make invalid calling sequence and run a group of stored procedures.

- Design several test sequences in which end users are likely to do business and do stress tests

2.3 Trigger tests

Test Coverage Criterion: *"EACH AND EVERY TRIGGER AND TRIGGER ERROR MUST BE TESTED AT LEAST ONCE"*

2.3.1 Updating triggers

Verify the following things and compare them with design specification

- Make sure trigger name spelling is correct
- See if a trigger is generated for a specific table column
- Trigger's update validation
- Update a record with a valid data
- Update a record, a trigger prevents, with invalid data and cover every trigger error
- Update a record when it is still referenced by a row in other table
- Make sure rolling back transactions when a failure occurs
- Find out any case in which a trigger is not supposed to roll back transactions

2.3.2 Inserting triggers

Verify the following things and compare them with design specification

- Make sure trigger name spelling

- See if a trigger is generated for a specific table column
- Trigger's insertion validation
- Insert a record with a valid data
- Insert a record, a trigger prevents, with invalid data and cover every trigger error
- Try to insert a record that already exists in a table
- Make sure rolling back transactions when an insertion failure occurs
- Find out any case in which a trigger should roll back transactions
- Find out any failure in which a trigger should not roll back transactions
- Conflicts between a trigger and a stored procedure/rules

  (i.e. a column allows NULL while a trigger doesn't)

2.3.3 Deleting triggers

Verify the following things and compare them with design specification

- Make sure trigger name spelling
- See if a trigger is generated for a specific table column
- Trigger's deletion validation
- Delete a record
- Delete a record when it is still referenced by a row in other table
- Every trigger error
- Try to delete a record that does not exists in a table
- Make sure rolling back transactions when a deletion fails
- Find out any case in which a trigger should roll back transactions
- Find out any failure in which a trigger should not roll back transactions
- Conflicts between a trigger and a stored procedure/rules

  (i.e. a column allows NULL while a trigger doesn't)

2.4 Integration tests of SQL server

Integration tests should be performed after the above component testing is done. It should call stored procedures intensively to select, update, insert and delete records in different tables and different sequences. The main purpose is to see any conflicts and incompatibility.

- Conflicts between schema and triggers
- Conflicts between stored procedures and schema
- Conflicts between stored procedures and triggers

2.5 Server setup scripts

Two cases must be tests: One is to set up databases from scratch and the other to set up databases when they already exist. Below is the minimum list of areas:

- Is a setup batch job available to run without much operator's assistance

(It is not acceptable if it requires an operator to run many batch jobs manually)

- Work environment the setup needs to run (DOS, NT)
- Environment variables (i.e. is %svr% defined?)
- Time it takes to set up
- Set up databases from scratch.
- Set up from existing databases
- Set up log and failure messages
- After setup, check for

Databases

Tables

Tables attachments (Keys, indexes, rules, defaults, column names and column types)

Triggers

Stored procedures

Look up data

User access privileges

3. FUNCTIONAL BACK END TESTS

Functional tests more focus on functionality and features of a back end. Test cases can be different from project to project. But many projects have things in common. The following section discusses the common areas. We encourage testers to add project-specific test cases in the functional test design.

3.1 How to divide back end on function basis

It is not a good idea to test a server database as a single entity at initial stage. We have to divide it into functional modules. If we cannot do the partition, either we do not know that project deep enough or the design is not modulized well. How to divide a server database is largely dependent on features of a particular project.

METHOD 1: We may ask ourselves what the features of a project are. For each major feature, pick up portion of schema, triggers and stored procedures that implement the function and make them into a functional group. Each group can be tested together. For example, the Forecast LRS project had four services: forecast, product lite, reporting, and system. This was the key for functional partitioning:

Figure 2. View a SQL server pertaining to the functionality

METHOD 2: If the border of functional groups in a back end is not obvious, we may watch data flow and see where we can check the data: Start from the front end. When a service has a request or saves data, some stored procedures will get called. The procedures will update some tables. Those stored procedures will be the place to start testing and those tables will be the place to check test results.

3.1 Test functions and features

Test Coverage Criterion: *"EACH AND EVERY FUNCTION OR FEATURE MUST BE TESTED AT LEAST ONCE"*

The following areas should be tested:

- Every feature no matter major or minor
- For updating functions, make sure data is updated following application rules
- For insertion functions, make sure data is inserted following application rules
- For deletion functions, make sure data is deleted correctly
- Think about if those functions make any sense to us. Find out nonsense, invalid logic, and any bugs.
- Check for malfunctioning
- Check for interoperations
- Error detection
- Error handling

- See if error messages are clear and right.
- Find out time-consuming features and provide suggestions to developers

3.2 Checking data integrity and consistency

This is a really important issue. If a project does not guarantee data integrity and consistency, we have obligation to ask for redesign. We have to check the minimum things below:

- Find out data protection mechanisms for a project and evaluate them to see if they are secure
- Data validation before insertion, updating and deletion.
- Triggers must be in place to validate reference table records
- Check major columns in each table and see if any weird data exist. (Nonprintable characters in name field, negative percentage, and negative number of PSS phone calls per month, empty product and so on)
- Generate inconsistent data and insert them into relevant tables and see if any failure occurs
- Try to insert a child data before inserting its parent's data.
- Try to delete a record that is still referenced by data in other table
- If a data in a table is updated, check whether other relevant data is updated as well.
- Make sure replicated servers or databases are on sync and contain consistent information

3.3 Login and user security

The following things need to be checked:

- Email validation
- SQL user login (user id, password, host name)
- NT server login
- Database access privilege (the sysusers table)

- Database security hierarchy
- Table access privilege (if 'select' is allowed.)
- Table data access control
- Training account (maybe no password is required)

There are more test cases here:

- Simulate front end login procedure and check if a user with correct login information can login
- Simulate front end login procedure and check if a user with incorrect login information fail to login
- Check concurrent logins (make many users login at the same time.)
- Try to login when a time-consuming query is running to see how long login will take to succeed
- Check for any security-restrict functions and see they are working properly
- See any data view restriction in place, such as, a user can see his data and the data of people who report to him.

3.4 Stress Testing

We should do stress tests on major functionality. Get a list of major back end functions/features for a project. Find out corresponding stored procedures and do the following things:

Test Coverage Criterion: *"EACH AND EVERY MAJOR FUNCTION OR FEATURE MUST BE INCLUDED IN STRESS TESTING"*

- Write test scripts to try those functions in random order but every function must be addressed at least once in a full cycle.
- Run test scripts over and over for a reasonable period
- Make sure log execution results and errors.
- Analyze log files and look for any deadlock, failure out of memory, data corruption, or nothing changed.

3.5 Test a back end via a front end

Sometimes back end bugs can be found by front end testing, specially data problem. The

following are minimum test cases:

- Make queries from a front end and issue the searches (It hits SELECT statements or query procedures in a back end)

- Pick up an existing record, change values in some fields and save the record. (It involves UPDATE statement or update stored procedures, update triggers.)

- Push FILE - NEW menu item or the NEW button in a front end window. Fill in information and save the record. (It involves INSERT statements or insertion stored procedures, deletion triggers.)

- Pick up an existing record, click on the DELETE or REMOVE button, and confirm the deletion. (It involves DELETE statement or deletion stored procedures, deletion triggers.)

- Repeat the first three test cases with invalid data and see how the back end handles them.

3.6 Benchmark testing

Test Coverage Criterion: *"EACH AND EVERY FUNCTION OR FEATURE MUST BE INCLUDED IN BENCHMARK TESTING"*

When a system does not have data problems or user interface bugs, system performance will get much attention. The bad system performance can be found in benchmark testing. Four issues must be included:

- System level performance

- Major functionality (Pick up most-likely-used functions/features)

- Timing and statistics (Minimal time, maximal time and average time)

- Access volume (A large number of machines and sessions must be involved.)

3.7 Common bugs

(To be filled in)

4. NIGHTLY DOWNLOADING AND DISTRIBUTION

This part is usually developed by an operation team. It did not get enough attention a long time

ago. However, after we deliver a product, end users must live with nightly process every daily. Bugs in nightly job put serious impact on users' daily work. Loss or corruption of customer data can be severe. We strongly recommend testers to intensively test nightly downloading and distribution, particularly error detection and reporting.

4.1 Batch jobs

By batch job, We mean batch jobs for Windows NT, DOS or OS/2. (The SQL scripts that are called by a batch job will be discussed in next three sections.)

Test Coverage Criterion: *"EACH AND EVERY BATCH JOB MUST BE TESTED AT LEAST ONCE"*

Here are the minimum list of test cases:

File transfer batch job:

- Destination path and source path
- All variables in a batch job must be resolved

(i.e. if %log% is used, the "log" must be defined somewhere either in the same file or in other system setup utility.)

- Make sure source files and their names are correct as specified.
- Make sure destination files and their names are correct as specified.
- Verify if any error level is checked after each file copy.
- Error messages must be logged. Fatal errors must be sent to operators or testers.
- Verify the database has the bulk copy option set to true before a batch job is executed
- Get estimate of total batch execution time. Make sure it fits the time window (specially the worst case)

BCP batch job:

- Make sure dbnmpipe.exe is automatically loaded and bcp.exe/isql.exe are on the system path
- Check for pass-in parameters %1, %2, ... to a batch job
- Make sure a table truncation script must be run before bcp in to those tables
- Make sure database name, bcp in file name/bcp out file name, and options /S, /U, /P, /c and /b
- Verify if any error level is checked after each bcp command.

- Failure should be logged. Fatal failure should be sent to operators or testers immediately

Batch file jobs that launch SQL scripts:

- Make sure dbnmpipe.exe is automatically loaded and isql.exe are on the system path
- Check for pass-in parameters %1, %2, ... to a batch job
- Make sure all required SQL files and their names, options /S, /U, /P
- Verify if any error level is checked after each launching
- Failure should be logged. Fatal failure should be sent to operators or testers immediately

4.2 Data downloading

In most cases, downloading is not just bcp in to a database. Data format change and calculations may happen.

Test Coverage Criterion: *"EACH AND EVERY SCRIPT MUST BE TESTED AT LEAST ONCE"*

We have to check the following areas:

- Network connection status. Failure handling.
- Input data must be validated before a batch job inserts/updates a database (Invalid data must be filtered out, i.e. NULLs in critical fields, negative values, too big numbers)
- Input data populated to right tables
- Calculations must follow business rules
- Check if data is really changed after data downloading
- See if two columns of data are mistakenly exchanged (i.e. product_id data and productgroup_id data are reversed when inserted)
- Check if any data is unexpectedly changed or deleted
- See what happens if database objects do not exist when downloading starts

4.3 Data conversion

The goals of many ITG projects are moving end users from existing VAX systems into PC platform systems. One of important steps is to convert old data into new systems. Data conversion is

required.

Test Coverage Criterion: *"EACH AND EVERY SCRIPT MUST BE TESTED AT LEAST ONCE"*

We list several checking items here:

- For each script, check for syntax error (Running a script is an easiest way to find out)
- For each script, check for table mapping, column mapping, and data type mapping
- Verify lookup data mapping
- Run each script when records do not exist in destination tables
- Run each script when records already exist in destination tables
- Make sure the execution sequence of scripts is correct

(i.e. Look up data must be converted first before conversions)

- Look for any scripts that encounter index error or trigger errors

(i.e. error attempt to insert unique index row.)

- Make sure a major transaction statement is followed by error checking "if @@error != 0"
- Look for any script that causes error out of memory at run time
- Check for any scripts that take too long to run for reasonable size of records. If it is the case, suggest developers to optimize scripts.
- Make sure "begin tran" and "commit tran" are in scripts
- If a failure occurs, transactions in a block after "begin tran" should be rolled back

4.4 Data distribution

There are two kinds of data distribution: One kind is to send replicated data to other SQL servers across LAN and WAN and keep them in sync. The other distribution is to pass information to other kinds of systems. Header information and data need converting. For example, KBATS article editing system nightly sends articles to Knowledge base server and to external system like CompuServe. Here is a minimum checking list:

Test Coverage Criterion: *"EACH AND EVERY FEATURE MUST BE TESTED AT LEAST ONCE"*

For data replication:

- Make sure every job extracts right updates from a SQL server
- Look for any new records that are missing in distribution data set
- Make sure data overwriting works correct
- Make sure sequences of data updates in destination servers
- Run distribution utility when new updates need to be distributed
- Run distribution utility when many changes are made
- Verify data loss handling mechanism for LAN or WAN environment
- Verify distribution mechanism when a network is down
- Verify error handling when a destination server is down
- Verify error handling when a source server is down
- Check if failures can be automatically recovered or can be recovered from next run

For data conversion distribution:

- Make sure every job extracts right data from a SQL server
- Look for any new records that are missing in distribution data set
- Make sure table mapping, column mapping, data type mapping, file format changes, and header changes
- Make sure data is converted to fit other systems

- Make sure data overwriting works correct
- Make sure sequences of data updates in destination servers
- Run distribution utility when new updates need to be distributed
- Run distribution utility when many changes are made
- Verify data loss handling mechanism in LAN or WAN
- Verify distribution mechanism when a network is down
- Verify error handling when a destination server is down

- Verify error handling when a source server is down
- Check if failures can be automatically recovered or can be recovered from next run

4.6 Common bugs

(To be filled in soon)

5. INTERFACES TO TRANSACTION APIS

By Transaction API, we mean those APIs that are specially designed for our projects to handle communications between a front end and a back end. Those APIs are not part of SQL DBLIB or ODBC. Although they do not belong to a back end, we have to test their interfaces to back end.

Figure 3. Connections between a front end and a back end

5.1 Connections to a SQL server database

- Make sure transaction APIs can open connections to a SQL server
- Verify APIs are able to send queries to a SQL server and retrieve data
- Unplug net cable and see if APIs can detect it
- Stop a SQL server and call APIs to make connection
- Stop a SQL server in the middle of transaction

5.2 Send queries to a back end

- Call each API that sends queries to a back end
- Make sure APIs call right stored procedures
- Verify parameters in stored procedure calls
- Make sure APIs should call stored procedures to access a SQL server. It is not recommended to send a simple query like "SELECT ... FROM ... WHERE..."

5.3 Receive output from a back end

- For every API, try a query with no row returned
- For every API, try more than one row returned
- For major API, try to have many rows returned
- Disconnect in the middle of transactions and check error detection
- Make sure an API always checks the return value of a stored procedure

- When a failure occurs, an API should receives value nonzero

5.4 Common bugs

(To be filled in soon)

6. OTHER TEST ISSUES

6.1 Test tips

- No program is bug free. If you have been doing tests for several days and do not find any bugs, our test methods or test data might be wrong. You should at least have some suggestions for developers.

- The Break-Program attitude is highly recommended. If you don't break it now, end users will break it later.

- There are a huge number of test cases. Always ask yourselves if any test case is missing and if our test specifications are complete.

- When you design test specification, think about valid cases, invalid cases and boundary cases

- Effective test methodology is neither unique nor universal. Feel free to discuss the test plan, test specifications and test data with other testers.

- When testing, you should pretend to be different levels of users. As "power" users, you can test advanced features heavily. As novices, you can do "stupid" things, i.e. turn off the machine in the middle of a transaction.

- If you suspect any result or message, go ahead and track it down. You may have found a "big" bug.

- Before you log a bug into Raid, find out the minimal steps to reproduce it. If you can not reproduce a bug, make a note and try it next time.

- If a developer resolves a major bug as "By-Design", but you think it is crucial to fix, try to convince the developer and your test lead.

- If you can track a bug down to a code level, do it. You will learn something new from bug tracking

- If a test is likely to be repeated later, automate it.

- A good programmer may not be a good tester. Be proud of your ability to find

bugs.

- You are an important part of product development. You help to ensure the high quality of ITG products.

6.2 Test tools

As mentioned earlier in this document, there are not many good tools for back end testing. But some utilities can be used.

- SQL language:

  Write test scripts to call stored procedures, retrieve data, insert/update/delete records. Most back end test work can be done with the SQL language

- NT SQL utilities:

DOS utilities like isql.exe, bcp.exe

Windows applications such as WinQuery, ISQL/w, SQL Administration, SQL Object Manager, SQL Client Configuration Utility

- MSAccess:

We may take advantage of MSAccess's tables, queries, forms, reports, macros and modules.

- Excel:

MSQuery and Q+E are useful for data validation

- Our own test tools:

Several test tools have been developed. For example, stored procedures to log passed/failed for each test and to present test statistics. Contact MinF for those tools.

6.2 Useful queries

To facilitate testing, we post some useful queries here. They are just something good to start with.

- Check for data devices and log devices:

sp_helpdb <database_name>

- Check for space used:

sp_spaceused <database_name>

- Get information about an object in a database:

sp_help <object_name>

where *<object_name>* can be a table name, trigger name, stored procedure name and so on.

- Get trigger code, procedure code, or view code, do:

sp_helptext <object_name>

- Find out who is on system, whose host name and other information:

sp_who

- Change database:

user <destination database_name>

- Find out existence of SQL objects by type:

*select * from sysobjects where type = "<type>"*

where <type> can be

U ------- User table

V ------- View

P ------- Stored procedure

TR ------ Trigger

- Count the numbers of records in individual user tables:

select "print '"+name+"'

select count(*) from "+name+"

go" from sysobjects where type = 'U'

Note: this statement does do count(). It outputs a script that does count.

- Generate invalid test data:

declare @customer_id int

/ Generate an invalid customer company id. */*

select @customer_id = MAX (customercompanyid) + 1 from customercompany

- Make a name unique and general using SUSER_NAME():

```
declare @companyname varchar(40)
  select @ companyname = SUSER_NAME() + "'S TEST COMPANY"
```

- The following code is to go through every record in a table and put a number after a company name:

```
declare @number int, @companyname varchar(40)
select @number = 1
  select @companyname = MIN(companyname) from customercompany
  /* Change companyname */
while @companyname != ""
  begin   /* Update a companyname */
update customercompany
  set companyname = companyname + convert(varchar, @number)
    where companyname = @companyname
      /* Pick up next companyname */
select @companyname = MIN(companyname)
      from customercompany
      where companyname > @companyname
end
```