Massive Stochastic Testing of SQL

Don Slutz
Microsoft Research
dslutz@Microsoft.com

Abstract

Deterministic testing of SQL database systems is human intensive and cannot adequately cover the SQL input domain. A system (RAGS), was built to stochastically generate valid SQL statements 1 million times faster than a human and execute them.

1 Testing SQL is Hard

Good test coverage for commercial SQL database systems is very hard. The *input domain*, all SQL statements, from any number of users, combined with all states of the database, is gigantic. It is also difficult to verify output for positive tests because the semantics of SQL are complicated.

Software engineering technology exists to predictably improve quality ([Bei90] for example). The techniques involve a software development process including unit tests and final system validation tests (to verify the absence of bugs). This process requires a substantial investment so commercial SQL vendors with tight schedules tend to use a more ad hoc process. The most popular method[1] is rapid development followed by test-repair cycles.

SQL test groups focus on deterministic testing to cover individual features of the language. Typical SQL test libraries contain tens of thousands of statements and require an estimated ½ person-hour per statement to compose. These test libraries cover an important, but tiny, fraction of the SQL input domain.

Large increases in test coverage must come from automating the generation of tests. This paper describes a method to rapidly create a very large number of SQL statements without human intervention. The SQL statements are generated stochastically (or 'randomly') which provides the speed as well as wider coverage of the input domain. The challenge is to distribute the SQL statements in useful regions of the input domain. If the distribution is adequate, stochastic testing has the advantage that the quality of the tests improves as the test size increases [TFW93].

A system called RAGS (Random Generation of SQL) was built to explore automated testing. RAGS is currently used by the Microsoft SQL Server [MSS98] testing group. This paper describes RAGS and some illustrative test results.

Figure 1 illustrates the test coverage problem. Customers use the hexagon, bugs are in the oval, and the test libraries cover the shaded circle.

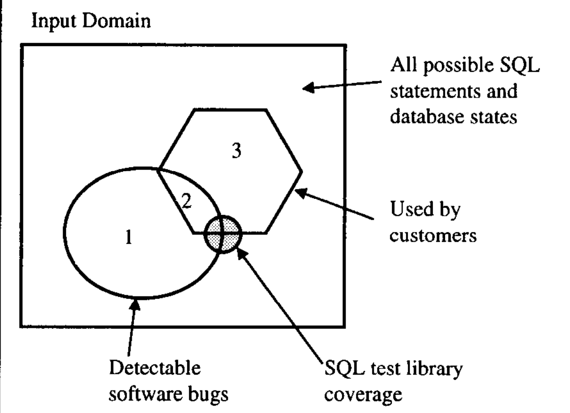

Input Domain
All possible SQL statements and database states
3
2
1
Used by customers
Detectable software bugs
SQL test library coverage

Figure 1:SQL test library coverage should include at least region 2. Unfortunately, we don't know the actual region boundaries.

2 The RAGS System

The RAGS approach is:

1. Greatly enlarged the shaded circle in Figure 1 by stochastic SQL statement generation.
2. Make all aspects of the generated SQL statements configurable.
3. Experiment with configurations to maximize the bug detection rate.

RAGS is an experiment to see how effective a million fold increase in the size of a SQL test library can be. It was necessary to add several features to increase the automation beyond SQL statement generation.

RAGS can be used to drive one SQL system and look for observable errors such as lost connections, compiler errors, execution errors, and system crashes. The output of successful Select statements can be saved for regression testing. If a SQL Select executes without errors, there is no easy method to validate the returned values by observing only the values, the

Proceedings of the 24th VLDB Conference New York, USA, 1998

[1] SQL testing procedures and bug counts are proprietary so there is little public information.

query, and the database state. Our approach is to execute the same query on multiple vendor's DBMSs and then compare the results. First, the number of rows returned is compared and then, to avoid sorts, a special checksum over all the column values in all the rows is compared. The comparison method only works for SQL statements that will execute on more than one vendor's database, such as entry level ANSI 92 compliant SQL[Ans92].

The RAGS system is shown in Figure 2 below. A configuration file identifies one or more SQL systems and the SQL features to generate. The configuration file has several parameters for stochastic SQL generation: the frequency of occurrence of different statements (Select, Insert...), limits (maximum number of tables in a join, maximum entries in a Group by list...), and frequency of occurrence of features (outer join, Where, Group by...). It also has execution parameters such as the maximum number of rows to fetch per query.

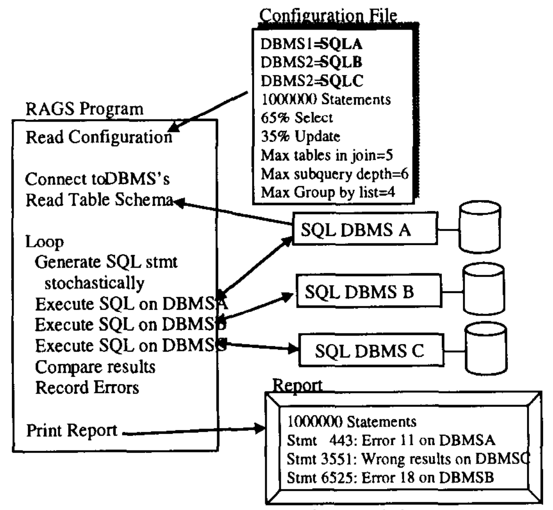


Figure 2: RAGS system. Several instances can be executed concurrently to represent multiple users.

The first step in running experiments on multiple systems is to ensure the databases are the same. They all must have identical schemas and identical data in their tables. It is not necessary that they have the same set of indexes or other physical attributes.

When the RAGS program is started, it first reads the configuration file. It then uses ODBC[MSO97] to connect to the first DBMS and read the schema information. RAGS loops to generate SQL statements and optionally execute them. Statement generation is described in the next section. If the statement is executed on more than one system, the execution results are compared. For numeric fields, the precision is reduced to a configurable value before the comparison is made. This avoids the problem of 1.999999 differing from 2.

At the end of the run, RAGS produces a report containing errors found, statistics of the run, and checksums of queries. A utility is provided that compares the reports from several runs and summarizes the differences. The comparison can be between different vendors or different versions of the same system (regression testing).

A typical SQL Select statement generated by RAGS is shown in Figure 3.

```
SELECT T0.au_id , LTRIM(('cuIe' +T0.au_id ))
FROM authors T0
WHERE NOT (NOT ((T0.au_fname )!= ANY (
 SELECT '}E'
 FROM discounts T1, authors T2
 WHERE NOT (('|K' )>= 'tKpc|AV' ) )) )
GROUP BY T0.au_id, T0.au_id
```

Figure 3. Select statement generated by RAGS.

The target database pertains to a publishing company. The stochastic nature of the statement is most evident in the unusual character constants and in unnecessary constructs such as "NOT NOT". RAGS also builds From lists, expressions, scalar functions, and subqueries stochastically but they appear less bizarre. Correlation names are used for tables to allow correlated column references. Constants are randomly generated (both length and content for character strings). RAGS uses parenthesis liberally, mostly to aid human recognition.

A somewhat larger RAGS generated SQL Select statement is shown in Figure 4 below. This type of statement is sufficiently complex that it is not likely to be found in a deterministic test library.

Our experience has been that about 50% of the Select statements return rows. This follows from the symmetry of predicates P and Not P occurring equally likely.

```
SELECT TOP 2 '6o' , ((-1 )%(-(-(T1.qty ))))/(-(-2 )), (2 )+(T0.min_lvl ),'_^p:'
FROM jobs T0, sales T1
WHERE ( ( (T0.job_id ) IS NOT NULL ) OR (('Feb 24 7014 10:47pm' )= (
 SELECT DISTINCT 'Jun  2 5147  6:17am'
 FROM employee T2, titleauthor T3, jobs T4
 WHERE ( T2.job_lvl  BETWEEN (3 ) AND (((-(T4.max_lvl ))%((3 )-(
 -5 )))-(((-1 )/(T4.job_id ))%((3 )%(4 )))) ) OR (EXISTS (
  SELECT DISTINCT TOP 7  MIN(LTRIM('Hqz6=14I' )),  LOWER( MIN(T5.country )),
   MAX(REVERSE((LTRIM(REVERSE(T5.city ))+ LOWER('Iirl' )))),  MIN(T5.city )
  FROM publishers T5
  WHERE EXISTS (
   SELECT (T6.country +T6.country ), 'rW' , LTRIM( MIN(T6.pub_id ))
   FROM publishers T6, roysched T7
   WHERE ( ( NOT (NOT (('2NPTd7s' ) IN ((LTRIM('DYQ=a' )+'4Jk`}A3oB' ), (
   'xFWU' +'6I6J:U~b' ), 'Q<D6_@s' , ( LOWER('B}^TK]`b' )+('' +'V;K2' )),
   '`min?' , 'v1=Jp2b@' )) ) ) AND (( EXISTS (
    SELECT TOP 10 T9.job_desc , -(-(T9.max_lvl )), '?[t\UGMNm'
    FROM authors T8, jobs T9, authors T10
    WHERE ( (T10.zip ) IS NULL ) OR (-((7 )%(-(1 ))) BETWEEN (-(((T9.job_id
    )*(-3.0 ))+(T9.min_lvl ))) AND (T9.min_lvl ) ) )
   ) AND (NOT (( (T7.hirange ) IN (T7.hirange , -(T7.hirange ), -(
   0 ), 1 , -(((-(-(T7.hirange )))/(-(T7.hirange )))-(T7.royalty )),
   T7.lorange )) OR ((-2.0 )< ALL (
    SELECT DISTINCT T8.hirange
    FROM roysched T8, stores T9, stores T10
    WHERE ( ( (1 )+((T8.royalty )%(-3 )) BETWEEN ((T8.hirange )*((T8.hirange
    )/(-4 ))) AND (T8.hirange ) ) OR (NOT (( (T8.royalty )= T8.hirange ) OR ((
    T8.hirange )< T8.lorange ) ) ) ) AND (T9.stor_id  BETWEEN (RTRIM(
    T8.title_id )) AND ('?' ) ) )
   ) ) ) ) ) AND ((( RADIANS(T7.royalty ))/(-3 ))= -2 )
   GROUP BY -(-((T7.lorange )+(T7.lorange ))), T7.hirange, T6.country
   HAVING -(COUNT ((1 )*(4 ))) BETWEEN (T7.hirange ) AND (-1.0 ) ) ) ))
)) AND (EXISTS (
 SELECT DISTINCT TOP 1 T1.ord_date , 'Jul 15 4792  4:16am'
 FROM discounts T2, discounts T3
 WHERE (T1.ord_date ) IN ('Apr  1 6681  1:42am' , 'Jul 10 5558  1:55Am' ,
 T1.ord_date )
 ORDER BY 2, 1 ) )
```

Figure 4: RAGS generated SQL Select statement for the publishing company database. The subqueries nest five deep and the inner queries reference correlated columns in the outer queries.

3 SQL Statement Generation

RAGS generates SQL statements by walking a stochastic parse tree and printing it out. Consider the SQL statement

```
SELECT name, salary + commission
FROM Employee
WHERE (salary > 10000) AND
  (department = 'sales')
```

and the parse tree for the statement shown below in Figure 5. Given the parse tree, you could imagine a

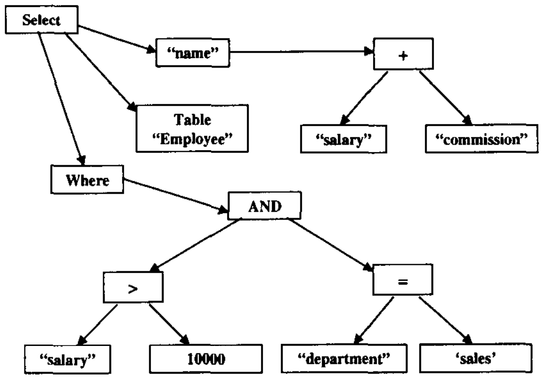


Figure 5: Parse tree for Select statement.

program that would walk the tree and print out the SQL text. RAGS is like that program except that it builds the tree stochastically as it walks it.

RAGS follows the semantic rules of SQL by carrying state information and directives on its walk down the tree and the results of stochastic outcomes as it walks up. For example, the datatype of an expression is carried down an expression tree and the name of a column reference that comprises an entire expression is carried up the tree.

RAGS makes all its stochastic decisions at the last possible moment. When it needs to make a decision, such as selecting an element for an expression, it first analyzes the current state and directives and assembles a set of choices. Then it makes a stochastic selection from among the set of choices and it updates the state information for subsequent calls.

On a 200Mhz Pentium RAGS can generate 833 moderate size SQL statements per second. The SQL statements average 12 lines and 550 bytes of text each. In one hour RAGS can generate 3 million different SQL statements - more than contained in the combined test libraries of all SQL vendors.

The starting random seed for a RAGS run can be specified in the configuration file. This allows a given run to be repeated without saving the SQL text. If the starting seed is not specified, RAGS obtains a seed by hashing the time of day.

4 Testing Experiences

This section contains examples of RAGS tests on a very small database (less that 4KB).

4.1 Multi-user Test

The results of a 10 concurrent user test are shown in Figure 6 below. Each user ran RAGS and generated a mix of Select, Insert, Update, and Delete statements.

Item	Number
Number of clients	**10**
Total number of statements	**25000**
Statements per transaction	**1 to 9**
Execution with no errors	**21518**
Errors expected:	
Deadlock victim	**2715**
Arithmetic error	**553**
Character value too long	**196**
Errors not expected (bugs)	
Error code 1	**13**
Error code 2	**5**

Figure 6. RAGS output for 10 clients executing 2500 statements each on one system.

Each of the 10 clients executed 2500 SQL statements in transactions that contained an average of 5 statements. Errors expected in random expressions include overflow and divide by zero. 86.1% of the statements executed without error, 13.8% had expected errors and 0.07% indicated possible bugs (18 occurrences of 2 different error codes).

4.2 Comparison Tests

The results of a comparison test between four systems are shown in Figure 7. The same 2000 random Select statements were run on each system. The numbers in each column reflect how that system's output compared to the output of the other three systems. The Comparison Case column enumerates the cases, with the dark circle representing the system of interest. The shaded ovals contain identical outputs. For example, the 15 in row 4 under system SYSB means that, for 15 statements, SYSB got the same output as one other system and the remaining two systems each got different outputs (or errors). Counts in row 5, where the specified system got a unique answer, are likely bugs.

4.3 Automatic Statement Simplification

When a RAGS generated statement caused an error, the debugging process was difficult if the statement was complex, such as in Figure 4. It was discovered that the offending statement could usually be vastly simplified by hand. The simplification involved removing as many elements of the statement as possible, while preserving the raising of the original error message (note that the simplified statement is not necessarily equivalent to the original statement).

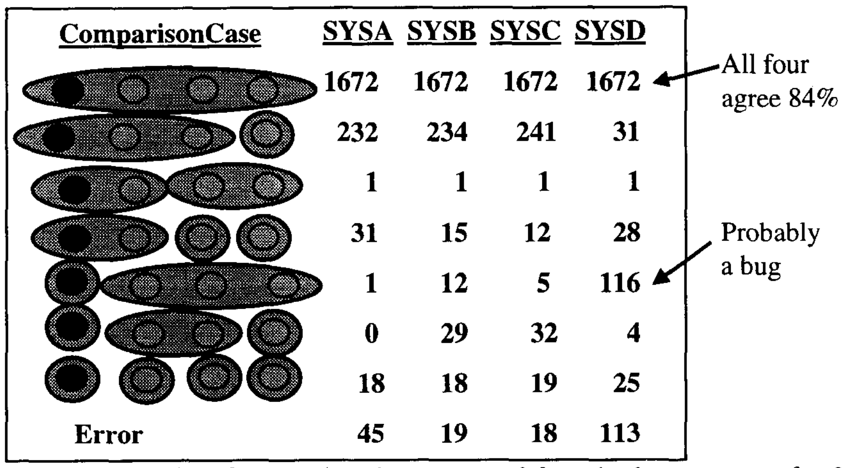
The simplification process itself was tedious so RAGS was extended to simplify the statement automatically. The RAGS simplified version of the statement in Figure 4 is shown in Figure 8.

To simplify a statement, RAGS walks a parse tree for the statement and tries to remove terms in expressions and certain clauses (Where and Having). This simplification algorithm was found to be very effective so it was not extended. For example, RAGS does not attempt to remove elements in the Select, Group by, or Order by lists

4.4 Visualization

To investigate the relationship between two metrics, such as statement execution times on two systems, a set of sample pairs is collected and analyzed.

RAGS presents an opportunity to scale up the size of such samples by several orders of magnitude. Not only does the scale up allow one to better analyze the relationship mathematically, it also allows one to plot


All four agree 84%
Probably a bug

ComparisonCase	SYSA	SYSB	SYSC	SYSD
	1672	1672	1672	1672
	232	234	241	31
	1	1	1	1
	31	15	12	28
	1	12	5	116
	0	29	32	4
	18	18	19	25
Error	45	19	18	113

Figure 7. Results of comparing the outputs of four database systems for 2000 Select statements. The numbers in row 5 indicate how many times this system got one result but the other three vendors all got a different result

the sample points and visualize the relationship.

```
SELECT TOP 2 '6o', -(-2), T0.min_lvl, '_^p:'
FROM jobs T0 , sales T1 WHERE EXISTS (
 SELECT DISTINCT TOP 1 T1.ord_date,  'Jul 15 4792  4:16am'
FROM discounts T2, discounts T3
ORDER BY 2,1)
```

Figure 8: RAGS simplified version of the statement in Figure 4. This statement causes the same error as the statement in Figure 4.

One example, shown in Figure 9, compares the execution times on two releases of the same system. With a few exceptions, the v2 release of SYSC is a little faster for the smaller queries and about the same for the larger ones.

5 Extensions

SQL coverage can be extended to more data types, more DDL, stored procedures, utilities, etc. The input domain can be extended to negative testing (injecting random errors in the generated SQL statements). Robustness tests can be performed by stochastically generating a whole family of equivalent SQL statements and comparing their outputs. Equivalent statements are obtained by permuting operands and lists (From and Group by) and adding useless terms (AND in a factor that is always TRUE). Testing with equivalent statements has the important advantage of a method to help validate the outputs.

In the performance area, the optimizer estimates of execution metrics, together with the measured execution metrics, can be compared for millions of SQL statements.

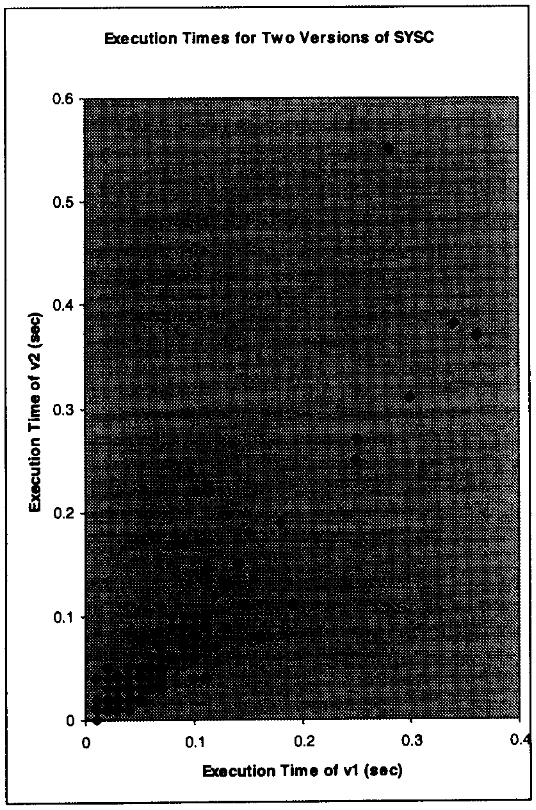


Figure 9: Relationship of 990 Select statement execution times on two versions of the same system. Version v2 is about as fast as version v1.

6 Summary

RAGS is an experiment in massive stochastic testing of SQL systems. Its main contribution is to generate entire SQL statements stochastically since this enables greater coverage of the SQL input domain as well as rapid test generation.

The problem of validating outputs remains a tough issue. Output comparisons for different vendor's database systems proved to be extremely useful, but only for the small set of common SQL The differences in NULL and character string handling and numeric type coercion in expressions was particularly problematic (these are also portability issues).

The outcome of our experiment was encouraging since RAGS could steadily generate errors in released SQL products.

References

[Ans92] ANSI X3.135-1992, American National Standard for Information Systems – Database Language – SQL, November, 1992.

[Bei90] B.Beizer, "Software Testing Techniques," Van Nostrand Reinhold, New York, Second Edition, 1990.

[MSO97] Microsoft ODBC 3.0 SDK and Programmer's Reference, Microsoft Press, February, 1997.

[TFW93] P.Thevenod-Fosse, H. Waeselynch, "Statemate applied to Statistical Software Testing," ISSTA '93, Proceedings of the 1993 International Symposium on Software Testing and analysis, pp 99-109.

[MSS98] Microsoft SQL Server Version 6.5, http://www.microsoft.com/sql.