Massive Stochastic Testing of SQL

Don Slutz

August 12, 1998

Technical Report
MSR-TR-98-21

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

A shorter form of this paper was accepted for VLDB 98 and appears in the Proceeding of the 24th VLDB Conference, New York, USA, 1998.

Massive Stochastic Testing of SQL

Don Slutz
Microsoft Research, 301 Howard St. #830, SF, CA 94105
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. This paper describes RAGS and the results from turning it lose on several commercial SQL systems.

1. Testing SQL is Hard

Good test coverage of 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 ([1] 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 ah-hoc process. The most popular method[1] is rapid development followed by test-repair cycles.

SQL test groups mainly use deterministic testing. They compose test scripts of SQL statements that cover individual features of the language and commonly used combinations of features. The scripts are continuously extended for added features and to verify bug fixes. If the test-repair cycles uncover particularly buggy areas, more detailed scripts for those areas are added.

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 minute, fraction of the SQL input domain. Stochastic testing can be used to increase the coverage. Stochastic techniques are used to mix several deterministic streams for concurrency tests and for scaled up load testing.

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 [2].

A system called RAGS (Random Generation of SQL) was built to explore automated testing. RAGS is currently used by the Microsoft SQL Server[3] testing group. This paper describes how RAGS works and how it was evolved to be a more effective tool. We focus on positive tests in this paper, but mention other kinds of tests in the summary.

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

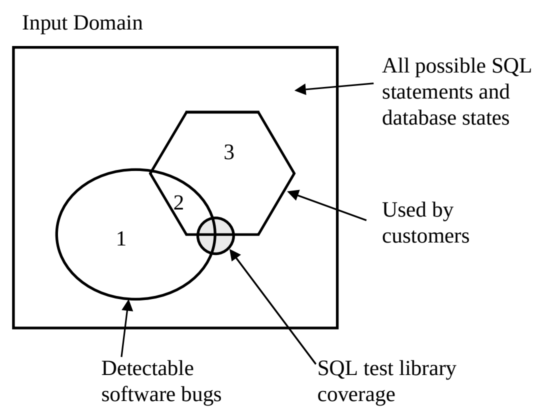


Figure 1:SQL test library coverage should include at least region 2. Unfortunately, we don't know the actual region boundaries.

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.

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

It is important to make all aspects of the generated SQL statement configurable so one can enlarge the reachable portion of the input domain and have a better shot at covering regions 1, 2, and 3 and beyond.

RAGS is an experiment to see how effective a million fold increase in the size of a SQL test library can be. RAGS was used on a number of commercial SQL systems that run on Microsoft NT. As RAGS was built and used, 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 query, and the database state. Our approach is to simply execute the same query on multiple vendors' 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[4].

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.

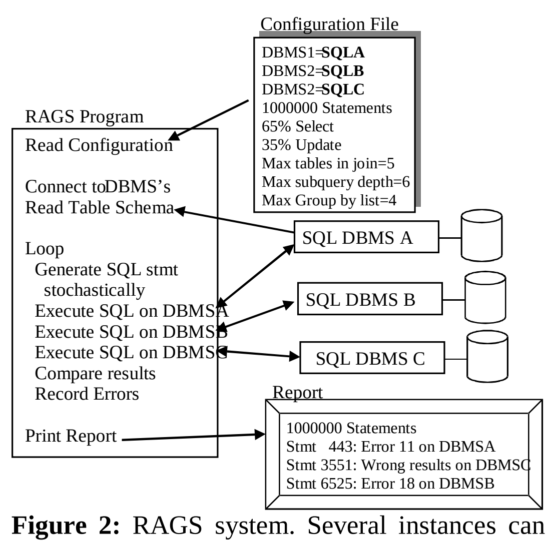


Figure 2: RAGS system. Several instances can be executed concurrently to represent multiple users.

The first step in running experiments on multiple systems is to ensure the databases are the same. They all must have identical schema 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[5] to connect to the first DBMS and read the schema information. When the schema information is read, columns with data types that RAGS does not support (or is configured to not support) are ignored. A table is ignored if all its columns were ignored. RAGS will optionally list the schema it is working against and the configuration settings in its output report file.

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 modification statements, the number of affected rows is compared[2]. For Select statements, the number of result rows and a special checksum are compared. The checksum is formed by building a checksum for each row and then combining the row checksums in an order independent manner. For numeric fields, the precision is reduced to a configurable value before the checksum is formed. This avoids the problem of 1.999999 differing from 2. Columns with datetime types are problematic when comparing results so we added a configuration parameter to preclude the use of timestamp functions in the generated SQL statements.

At the end of the run, RAGS produces a report containing errors found and statistics of the run. If several RAGS programs are run concurrently, a utility produces a reduced report that lists each distinct error code together with a count of the number of occurrences.

RAGS can also be configured to run on just one database and include the row counts and checksums in the output report. The outputs from different runs at

[2] The database states are also compared at the end of the run.

different times can be compared to find discrepancies. This is useful for regression tests.

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

```
SELECT T0.au_id , LTRIM((’cuIfzce’
+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 that has tables for authors, books, stores, sales…. 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.

One might wonder how often stochastic Selects actually return rows of data. Our experience has been that about 50% of the non-error causing Select statements return rows while the remainder return no rows. The maximum join size together with the size of the database influences how many rows are returned. For large databases, it is important to timeout long running statements or keep the maximum join size and the maximum subquery depth quite low to preclude excessive run times or encountering system limits.

```
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

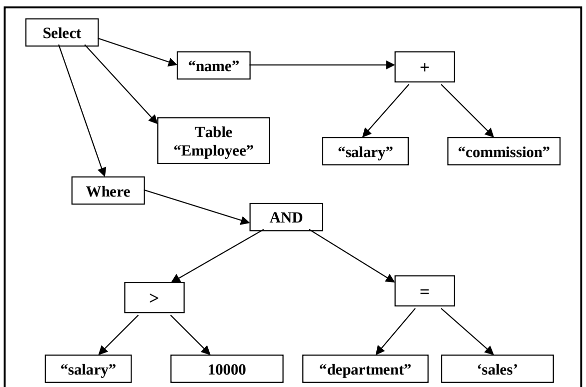


Figure 5: Parse tree for Select statement.

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 takes a couple of detours, like generating the tables in the From list and the columns in the Group by list, before it begins outputting but it essentially make one pass of the 'tree'.

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 and being inside an aggregate expression, are carried down the 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 and with moderate configuration (maximum join size 3, maximum subquery depth of 5, maximum expression depth of 6), RAGS can generate 833 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 of average size - more than contained in the combined test libraries of all SQL vendors.

For limits testing, the sizes of expressions and predicates can be increased so that typical expressions or predicates contain hundreds of terms and span several pages. The lengths of Select, Group by, and Order by lists can also be increased. For concurrency tests, the maximum join size, subquery depth, and expression depth are set very low and the relative occurrence of Insert, Update, and Delete statements increased to get lots of small, simple data manipulation statements.

Each SQL statement can be characterized by the configuration settings, the database schema, and the seed of the random number generator when statement generation began. For the same configuration and schema, if the same seed is supplied on a different run, the same statement will be generated.

If RAGS is configured to execute the SQL statements and an error is found, RAGS records the seed for the statement along with the error message. If the statement runs without error, an output signature, consisting of the number of rows modified or fetched, together with the checksum for Selects can optionally be recorded for later comparisons with other systems or for regression testing.

The starting seed for an entire RAGS run can be specified in the configuration file, allowing a given run to be repeated - it is not necessary to save the SQL text in the test library. If the starting seed is not specified, RAGS obtains a seed by hashing the time of day.

4. Testing Experiences

The evolution of RAGS is driven by the needs of users. Many of the changes made do not deal with the stochastic SQL statement generation, but increase the degree of automation. Since the goal of RAGS is to run many orders of magnitude more statements through the system, if even a small percentage of the statements needs human attention, the total effort can be prohibitive. As each change to RAGS was made, more items were added to the configuration file which now contains almost 100 items.

This section contains examples of RAGS output and enhancements made to improve usability. Tests were performed on systems from different vendors in order to validate output. To avoid specific vendor comparisons, vendor identities are suppressed. The systems are referred to as SYSA, SYSB, etc. and the mapping to actual vendors is changed from test to test. Additionally, the total size of the test database is less that 4KB, sufficiently small to preclude meaningful comparisons.

Multi-user Test

The output of a multi-user test on one system is shown in Figure 6 below. This was a concurrency test and used 10 clients, each generating and executing small Select, Insert, Update, and Delete statements on a single database system.

Item	**Number**
Number of clients	**10**
SQL statements per client	**2500**
Total number of statements	**25000**
Statements per transaction	**1 to 9**
Execution Results:	
Number of Selects	**6401**
Number of Inserts	**6165**
Number of Deletes	**6221**
Number of Updates	**6213**
Number of Transactions	**4997**
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**

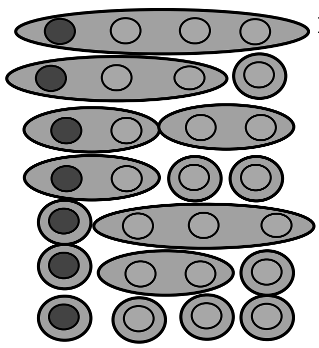
Figure 6. RAGS output for 10 clients executing 2500 statements each on one system.

Each of the 10 clients executed 2500 SQL statements in transactions that contained an average of 5 statements. Error counts are the number of times ODBC calls returned an error code. The errors are partitioned into errors expected from stochastic statements and those that are likely bugs. Expected errors include deadlock (with multi-user), arithmetic (overflow and divide by zero in random expressions), and character values too long (too many concatenations of fixed length values for insert or update). 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).

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 they got errors). The first row has the counts where all 4 systems agreed and the second row has the counts where the system agreed with two other systems.



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

All four agree 84%

Probably a bug

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

Counts in the fifth row, where the specified system got one answer and the other three systems all got a different answer (and all three agreed) is a great place to look for a bug. The fifth row also represents the situation where one system just has different behavior than the others (see the Validating Output section below).

Database Connections

RAGS uses ODBC to connect to a database and it reads the schema information directly using the SQLTables and SQLColumns calls of the ODBC API. The connection parameters for each type of vendor were added to the configuration file. We found that each vendor required a different mix of configuration options so these were added too.

If a database connection is lost, RAGS trys to reconnect. If the reconnect fails it assumes the server is down and it aborts the run. If crashes occur, it is necessary to automate a rapid restart of the database server. The restarts were not done within RAGS, instead they were done in the macros that call RAGS to run the tests.

Transactions

When Insert, Update, and Delete statements were added to RAGS, the ability to generate transactions was also added. The maximum length of a transaction can be configured. If the maximum is set greater than 0, RAGS will pick the number of statements (not counting the Commit or Rollback) in each transaction uniformly between 1 and the maximum number. If the maximum is set to 0, the Auto Commit feature of ODBC is used and each statement is a separate, committed transaction.

When transactions are enabled, the percentage of transaction that commit is configurable (the remaining transactions are rolled back). An interesting multi-user experiment is to set the percentage of commits to 0 and compare the database state before and after the runs (it should be the same since all transactions are rolled back).

Automatic Statement Simplification

When a RAGS generated statement caused an error, the debugging process was often difficult if the statement was long and complex, such as the example in Figure 4. It was discovered that the offending statement could usually be vastly simplified by hand. The simplification involved taking out as many elements of the statement as possible, while preserving the raising of the original error message. Note that the simplification process does not produce a statement that is equivalent to the original SQL statement. The simplified statement just results in the same error message (the error may not even be caused by the same bug, but that is unlikely).

The simplification process itself was very tedious so RAGS was extended to optionally simplify the statement automatically. The statement in Figure 4 was one that caused a compiler error in one of the database systems. The RAGS simplified version of the statement is shown in Figure 8.

```
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.

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). Rags blanks out the expression or clause and reexecutes the statement. If the same error occurs, it continues its walk. If a different error or no error occurs, it replaces the blanked out text and continues. We found this simplification algorithm to be very effective so we did not extend the algorithm to be more complex. For example, RAGS does not attempt to remove elements in the Select, Group by, or Order by lists and it never backtracks on its walk to retry simplifications in a different order.

Error Filtering

An early version of RAGS would echo any error messages it found during statement execution in the output file. With long runs the output files became huge because the RAGS caused errors such as divide by zero and arithmetic overflow were numerous. RAGS was changed to simply count the number of these errors and only report the error text once. However, the bloated output file problem still occurred. When a configuration setting was found that caused errors from probable bugs, there would usually be lots of occurrences of the same errors in long runs. The solution was to extend the configuration file to include a list of error codes to ignore.

Validating Output

To validate the output of successful queries, the queries are executed on database systems from different vendors and the outputs are compared. Our first step was to add a configuration flag to RAGS to restrict the generated SQL to be compliant with ODBC Core SQL (a subset of entry level ANSI 92 SQL). It was found that the CORE subset of SQL was too small and excluded many important features in most SQL implementations. So features such as built-in functions were added.

As the ANSI subset we supported was enlarged we quickly found several visible differences among vendor's ODBC and SQL implementations. Examples of different behavior include: treating empty character strings as NULL, producing a NULL value by concatenating a character string with a NULL, and the number of rows returned from aggregating over an empty grouped table. Configuration options were added to RAGS to prevent construction of some of the SQL statements containing elements where vendors had different behavior. It was quite challenging to get a diverse set of SQL statements to run on several different systems.

It eventually became necessary to specialize a RAGS run to the target SQL vendor so the configuration file grew to specify the type of SQL database system. This actually increased the automation significantly because we could quickly deal with problems without shrinking the domain of generated statements. For example, when execution by one vendor produced errors because of the default return types of scalar functions, we simply changed RAGS to cast the output to a different type for that vendor.

Visualization

When one wants to investigate the relationship between two metrics, such as the statement execution times of two systems, a set of sample pairs (execution times on both systems for a set of statements) is collected and analyzed. One might compute statistics such as the average of the ratio of execution times on both systems.

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 and also allows one to plot the sample points and visualize the relationship.

In one experiment the execution times for 931 Select statements was measured for systems SYSB and SYSD. The systems ran on different computers and the RAGS client was local for one run and remote for the other (thus, this is not a fair comparison). The average execution time of a statement on SYSD was 2.52 times that on system SYSB. The execution time on SYSD varied between 1/6 to 13 times that of SYSB. More interestingly, Figure 9 below shows a scatter plot of the execution times on the two systems. One can see that system SYSB is faster, except for the band of points at the bottom of the figure.

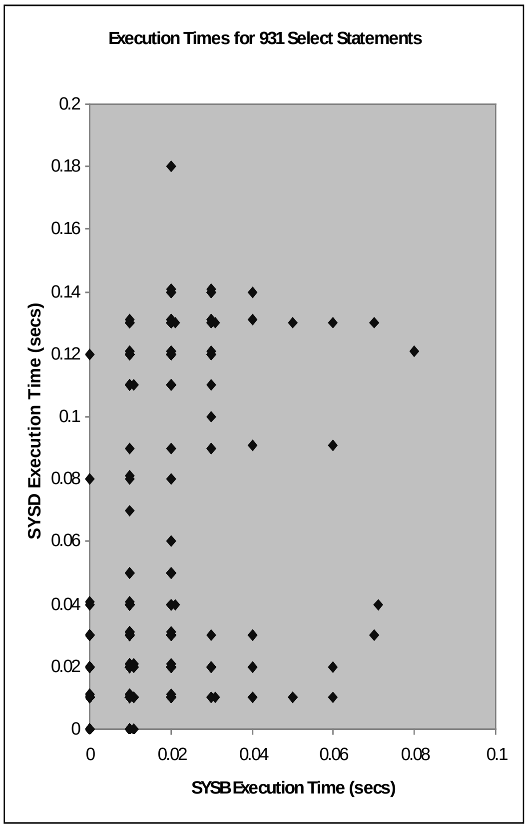


Figure 9: Relationship of execution times on two systems. The database is very small and the Selects are not complex.

Another example, shown in Figure10, compares the execution times of 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.

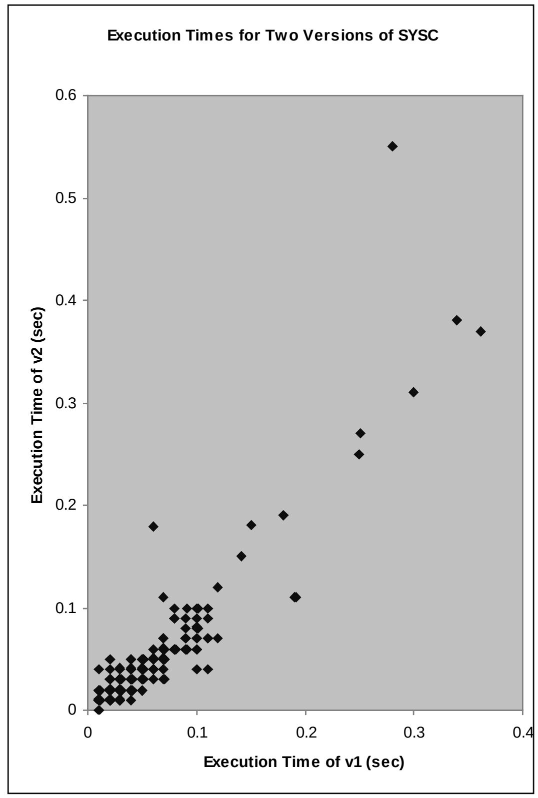


Figure 10: Relationship of 990 Select statement execution times on two versions of the same system. Version v2 is about as fast as version v1.

5. Extensions

RAGS is a project in progress. SQL coverage needs to be extended to more data types, more DDL, stored procedures, utilities, etc. Beyond that there are several possibilities for future work in stochastic SQL testing. First is a study of the effectiveness of this experiment: measure code coverage under random SQL testing and investigate bugs that customers see, but were not discovered by RAGS.

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 some operator operands and lists (From and Group by) and adding useless terms (AND in a TRUE term or multiply by a factor that evaluates to 1). Testing with equivalent statements has the important advantage of a method to 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. Although we avoided vendor performance comparisons in this paper, a more structured experiment, using a large random workload, would be interesting.

6. Summary

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

The problem of validating output remains a tough issue. The use of database systems from different vendors for output validation proved to be extremely useful for the SQL common to many vendors. The down side is that the common SQL subset is relatively small and changes with each release. We also found the differences in NULL and character string handling and numeric type coercion in expressions to be particularly problematic (these are also portability issues).

The outcome of our experiment is encouraging since we found that RAGS can steadily generate errors in released SQL products. To make RAGS effective it was necessary to automate other steps of the testing process, notably positive result comparisons and statement simplification for error cases.

6. References

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

[2] 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.

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

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

[5] Microsoft ODBC 3.0 SDK and Programmer’s Reference, Microsoft Press, February, 1997.