A Test-Suite Generator for Database Systems

Ariel Hamlin
MIT Lincoln Laboratory
ariel.hamlin@ll.mit.edu

Jonathan Herzog[ϕ]
Jonathan Herzog Consulting
jherzog@jonathanherzog.com

***Abstract*—In this paper, we describe the SPAR Test Suite Generator (STSG), a new test-suite generator for SQL style database systems. This tool produced an entire test suite (data, queries, and ground-truth answers) as a unit and in response to a user's specification. Thus, database evaluators could use this tool to craft test suites for particular aspects of a specific database system. The inclusion of ground-truth answers in the produced test suite, furthermore, allowed this tool to support both benchmarking (at various scales) *and* correctness-checking in a repeatable way. Lastly, the test-suite generator of this document was extensively profiled and optimized, and was designed for test-time agility.**

***Index Terms*—Database Testing & Benchmarking, Test Suite Generator, Data Generation, Query Generation**

I. Introduction

In this paper, we describe the SPAR Test Suite Generator (STSG), a new test-suite generator for SQL style database systems. Given a user's specification, the tool produced an entire test suite (data, queries, and ground truth results) for evaluating databases. This allowed users to create test suites individually crafted for a *specific* database systems being evaluated. Evaluation took the form of either benchmarking (at various scales) *or* correctness checking due to the inclusion of ground-truth answers to the generated queries in the produced test suite. The tool was written in Python [1], a high-level and object-oriented language which enabled quick development and test-time agility. Despite this, the test-suite generator of this document has been extensively profiled and optimized, and performs at a level comparable with other data-generating tools. We believe this software is of use for the database community as a testing aid. As such, along with the rest of the testing framework, we will open source the code.

Before we explain the design of the test-suite generator, it is useful to describe the project under which this tool was written and the requirements the tool had to meet.

[ϕ]Work was done while affiliated with MIT Lincoln Laboratory

This work is sponsored by the Intelligence Advanced Research Projects Activity under Air Force Contract FA8721-05-C-002. Opinions, interpretations, recommendations and conclusions are those of the authors and are not necessarily endorsed by the United States Government.

A. The SPAR project and requirements

The Security and Privacy Assurance Research program, or SPAR, was created in 2010 by the Intelligence Advanced Research Projects Activity (IARPA). The goal of SPAR was to design and build new privacy-preserving data searching technologies that are secure, fast, and expressive enough to use in practice. Though the project spanned many areas, this paper focuses specifically on the area of privacy-preserving database systems.

The database systems developed in this area were required to provide a number of security properties [2], which we do not consider here. Of more interest, however, they were required to provide a number of performance and functionality properties. For example, these systems were required to run reasonably fast: be within a factor of 10 of a non-privacy-preserving baseline (e.g., MySQL or MariaDB). They were required to scale to 100M rows and 10TB of total data. They were required to support a variety of data types (integers, enums, free-text fields, etc.). Finally, they were required to support a subset of the following query-types (of their choice):

- EQ: Straightforward `field = value` matches.
- BOOL: Boolean expressions (`AND` or `OR`), with up to 6 leaf-clauses.
- RNG: Range queries (less-than, greater-than, and between) over over numeric, strings and enumerated data types.
- KWD: Free keyword search over text fields.
- STM: Keyword search with stemming over text fields.
- WILD: String-match with wild-cards.
- SUB: Sub-string matching.
- THR: M-of-N thresholds.
- RANK: Threshold queries (THR) ranked by the number of matched terms.
- PROX: Close proximity of two given keywords in free-text fields.
- XML: Keyword searches on XML-formatted data.

Due to the privacy requirements, correctness in the performer's systems was not a trivial problem to solve. They were required to have no false negatives and only a very small false positive rate in the records they returned.

To evaluate these new database systems, MIT Lincoln Laboratory acted as the Test and Evaluation (T&E) team

978-1-4799-6233-4/14/$31.00

and measured the correctness and performance of each new database system developed under the project. To do this, we needed the ability to generate performance-tests and correctness-tests for a variety of query types and over two dimensions:

1) The size of the database, which needed to range between 1K and 100M rows, and
2) The number of rows returned by a given query, which ranged from zero to a large fraction of the database.

Rows and queries needed to be realistic to illustrate an actual use case with privacy concerns (so that our evaluations could be appreciated by potential government customers). Furthermore, though we would be given time to prepare the tests, actual testing would be limited to one week per performer system. Thus, we needed a high degree of test-time agility so as to recover from test-time problems quickly.

Also, we were required by IARPA to provide our own test apparatus to the performers well in advance of testing periods (for debugging purposes). This included a test harness, automatic report generator, and the test suite generator [3]. We needed to ensure that the same set of tests would be given to each system and that the tests could be repeatable. Hence, the test-generator needed to be *pseudo-random*: deterministic when given a top-level random 'seed', but unpredictable without it.

B. Prior work

We examined other test-suites and data-generators for database systems, but found that none of these other tools met the team's needs.

The Flexible Database Generators [4] can construct a database from a set of table schema with associated distributions. However, we needed distributions beyond those supported, guaranteed repeatability, and ground truth. Due to the limited testing period, moreover, we needed the ability to quickly generate new queries of any given type (and associated ground truth) for the data already in a database under test.

MUDD [5] seemed to meet our data-generation needs, but did not support query-generation. Likewise, Targeted Queries for Database Testing [6] could generate many of our query types, but not all. RAGS [7] could neither produce queries of all the needed types nor generate the data itself.

The DBGen and QGen system [8] were both closer but there were a number of mismatches between these tools and our needs. The schema of the DBGen system is effectively hard-coded, and we needed finer-grained control over the data produced (e.g., row size) than it could support. Also, we needed to generate queries not among those supported by the QGen system, and the ability to specify the (approximate) number of rows returned by a given query. Lastly, they were missing one major component: ground-truth results for generated queries.

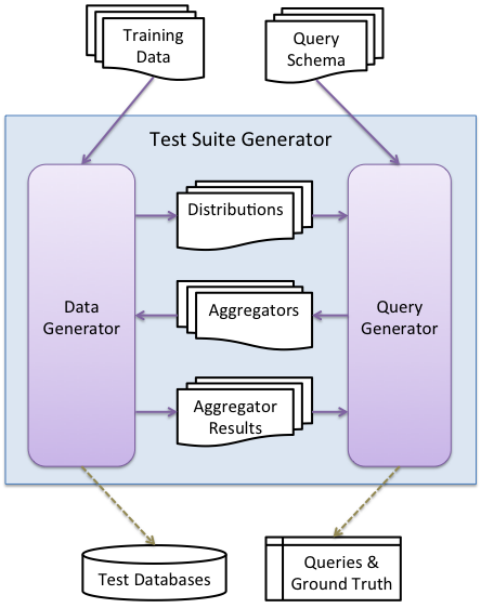

Figure 1: The architecture of the test suite generator.

II. Data Generator

Our data generator component generated synthetic people: one person per row. More specifically, each row contained: demographic data, such as name (string), date of birth (date), state of residence (enum), income (integer), etc.; long fields English-language text[1] (for KWD and STM searches); XML-formatted data; and a 100KB 'fingerprint' field (binary data) to ensure that our databases met the project's size requirements.

To create these rows, the data generator took three inputs: a master seed used to initialize randomness, a file specifying which fields were needed, and the dataset used to train the data generator. For the SPAR project, our dataset consisted of files from the US Census Bureau and Project Gutenberg. The Census Bureau publishes a wide variety of data suitable for this purpose, such as lists of the 10,000 most common first and last names (with weights) [9], anonymized demographic data for 15M representative Americans (with weights) [10] and so on. Project Guteberg also publishes a number of public-domain texts (e.g., novels) that we used as a sample of written English text.

From these datasets, the data-generator extracted a probability-distribution object for each field to be generated. With one exception, these underlying distributions were not standard distributions but instead exactly fit the

[1] ranging from 20 to 10,000 bytes

training data. Given the size of the training corpus, we believed exact-fit distributions to be sufficiently realistic for testing purposes. The only distribution which was not exact-fit was that for English text, which uses a trigram model of written English. In these models, the probability of each token (word or punctuation) was conditioned on the previous two.

Also, our analysis of the training data indicated that without loss of realism, most fields could be given independent distributions. Data-realism did require some dependencies, however. For example, it was determined that 'marital status' depended heavily upon on 'age' and 'military service' in the original data set. In these cases, the probability distribution for the dependent variable was conditioned on the values selected for the independent variables.

The final output of training was a single, top-level 'row distribution' object with a slightly surprising interface. Instead of a single 'generate row' method, it actually provided: a set of distribution objects, each representing one field of the row; and a list indicating the order in which these distributions are to be called.

This list served two purposes. First, it provided a field-generation order that was guaranteed to generate independent variables before the relevant dependent variables. (Thus, the calls to conditional distributions are well-founded.) Secondly, it could be used for a simple optimization: automatically omitting unneeded fields. Because the data-generator was given an input file specifying which fields were actually needed, it could *stop* generating values when all specified fields had been chosen. If the user needed only inexpensive values, then this optimization could become very significant in practice.[2]

The process of distribution-extraction was extremely fast: on the order of 15 minutes. And once generated, the row-distribution object was used to generate rows in parallel. This parallelization was crucial to meeting our performance requirements: we needed a system that scaled with multiple processes in parallel in order to generate the larger databases. This should have been straightforward, but limitations of the Python `multiprocessing` module prevented us from using the natural built-construct (`Pool`). Instead, we implemented our own. This implementation used a central 'mothership' to dispatch row-generation jobs to 'workers'. The SPAR requirements dictated two interesting aspects of this design. First (and as is the case with `multiprocessing.Pool`) performance was improved by dispatching the rows to workers in multi-row batches: the larger the better. Second, the reproducibility requirement dictated that the workers' output needed to be completely deterministic based on the master seed value. The mothership, before sending off workers to generate a batch of rows, generated the set of row IDs for each row in the batch and a corresponding randomness seed for each. The worker then re-seeded its own internal random-number generator with the given seed for that row before generating the values. This means that each row ID corresponded to exactly the same row regardless of how the mothership dispatched rows to the workers.

One last, key aspect of the data generator was its support of map-reduce [11] over the rows being generated. This served two purposes: to support a variety of output formats, and to provide query-generation a mechanism by which to collect ground-truth results. Along with row-batches and randomness seeds, workers were also given a set of map-reduce jobs, or *aggregators*. As they generated rows, the workers used these jobs to maintain a running aggregate value (one for each job) over all rows generated so far. At the end of generation, the workers returned these aggregate values to the mothership who then used the same map-reduce jobs on them to produce a single overall, final value. We note that output formats could be (and were) encoded using very simple aggregators. Such aggregators implemented row-output in the `map` method, and left the `reduce` method as a no-op. On the SPAR project, we implemented aggregators of this type for both a flat file format and insertion into an SQL database. We also note that this paradigm could be used to extract aggregate statistics such as averages or histogram counts.

III. Query Generation

The goal of the query generator was two-fold: to automatically and repeatably generate SQL-style queries for a variety of query types, and to provide ground truth results for those queries. Although the data generator ran in a stand-alone fashion, query generation needed to be run in conjunction with data generation. The query-generator proceeded through three main phases, each of which required interaction with the data-generator (see Figure 1):

1) Generation of the queries, which required the distribution objects from training,
2) Collection of ground truth, which used the map-reduce framework, and
3) Refinement of the queries, which required the ground truth collected.

The query generator took one input: a configuration file which specified the queries needed for the test suite. At a high level, this specification was a list of (number of queries, query type, min, max...) tuples, where each tuple indicated that the generator is to return 'number of queries' of 'query type' that matched between 'min' and 'max' rows in the database. For example, the triple (10, EQ, 100, 1000) indicates that we desired 10 EQ queries where each query matches between 100 and 1000 rows.

[2] As we discuss in Section IV, the text fields were most expensive to generate. Thus, we ensured that they would be at the end of the list.

However, many query types required additional parameters. This included: which fields to generate queries for; number of clauses in BOOL, RANK, and THR queries; and many other properties for different query types.

For readability, the number of records to match (i.e., the upper and lower bounds 'min' and 'max' above) were expressed in absolute terms, not percentages of the database. However, the query generator transformed the absolute bounds into normalized frequencies by dividing the bounds by the number of rows being generated (obtained from the data generator). In many cases, this made it easy for the query-generator to create simple queries for a given field: it randomly chose a weight from the appropriate range, and asked the data-generator's probability distribution for an appropriately weighted value.

Compound queries, on the other hand, required a different process. We first created lists of fields that were dependent upon each other, and selected a set of dependent fields that had both the correct number of fields (i.e. greater than or equal to the number of clauses) and appropriately sized supports.[3] From that set, if the final query needed to have n clauses, the query generator would select n fields (without replacement). It then would compute individual weights for each field in a way that depended on the compound query type. For example, if the top-level weight for a BOOL conjunctive query was to be x, then weights for individual clauses were calculated as $x^{1/n}$. Similar formulae were used for the other conjunctive queries.

Ideally it would be possible to select single values for each field with the right characteristics to satisfy the compound query requirements. Due to the messy nature of real world data, though, it was necessary to over sample values. That is, the query generator would select *multiple* values for each field, where each value had approximately the right weight. In this way, it could pick and choose from the selected values during refinement (discussed below).

To collect ground-truth results for each query (and each clause of compound queries), the query was given to an aggregator. This aggregator was parameterized by information about the query: field value, targeted result set size, keyword length, etc. Furthermore, the `map()` function would return the ID of the row if the row matched the underlying query, and an empty list otherwise, and the `reduce()` function, when given two lists, would append the two lists. In this way, the aggregator of a given query would build a list of all rows that matched the query in question.

Once initialized, aggregators were then given to the map-reduce framework of the data-generator to collect the IDs of matching rows. The query generator then used these ground-truth answers for refinement of the queries. For simple queries, refinement was fairly straightforward: the query generator simply ensured that the number of rows actually matched by a given query fell within the range specified by the user. For more complicated queries, however, the query-generator used a semi-intelligent search method to find combinations of clauses that matched all required metrics (matching results set size, number of results for the first clause, etc). Though effective, this approach affected performance. (See Section IV-B.)

Once all refinement was finished, the results were output in two different ways: the `WHERE` clauses were written to a text file, and the full results (`WHERE` clauses, metadata and ground truth information) were written to a SQLite database. This SQLite database could then be used by the rest of the test harness developed by the SPAR program. (see [3] for more details.)

IV. Performance

In order to allow fast re-executions of the test suite generation during testing, we invested significant effort into optimizing the efficiency of our tool. In this section, we will first describe the optimizations we made, and then provide the final performance numbers.

A. Optimizations

We look first at those optimizations in data generation, and then move to those of the query generator. In the data generator, optimizations came (mostly) in two flavors: hoisting[4], and pushing execution from Python to C. For example, one bottleneck was text-generation. Although training's text-corpus contained only 1.1 million tokens, it was used to generate 180 *billion* tokens during data-generation. Therefore, it paid great dividends to move work from text-generation to text-ingestion. For example, the 'generate next token' function was originally written to test (multiple times) whether the generated 'token' was a sentence-ending token such as a period. By moving this test to text-ingestion, we were able to eliminate the need for this test in 'generate next token'. This, and similar hoisting, reduced data-generation's execution time by approximately 35%.

In the end, though, the 'generate next token' function was still the most frequently-called method of our code by a factor of 7.5. Therefore, we moved this computation from Python to C using the Python-to-C translator Cython [12]. By adding a only few type annotations, we enabled Cython to produce extremely fast monomorphic C code that reduced data-generation time by another 40%.

[3] 'Appropriately sized' depended on the type of compound query.

[4] Hoisting (or loop-invariant code motion) in this case involved removing calculations that did not change during data generation and placing them within the training process so they would only be executed once rather than once per row.

```
for (id, randomness_seed) in batch:
    row = generate_row(id, randomness_seed)
    for job in map_reduce_jobs:
        map_val = job.map(row)
        agg_val[job] = \
            job.reduce(map_val, agg_val[job])
```

```
rows = [generate_row(id, randomness_seed)
        for (id, randomness_seed) in batch]
for job in map_reduce_jobs:
    map_vals = map(job.map, rows)
    agg_val[job] = \
        reduce(job.reduce, map_vals)
```

Figure 2: Imperative vs. functional forms of our map-reduce

A second set of data-generation bottlenecks arose in the map-reduce framework. In this case, however, the problem was not the innermost function but the top-level loop. In particular, the top level of our map-reduce application was originally written in an imperative style. To optimize it, we re-wrote it into a functional style (Figure 2). This simple change applied both of our previous optimization strategies. First, it hoisted the work of function-resolution. For example, the imperative style required the method `job.map` to be resolved once per job per row, while the functional style required it to be resolved only once per job. (Recall that we were generating 100M rows, so this saves quite a bit of redundant work.) Also, the `for` loops of our imperative code were being executed in interpreted Python, while the use of the `map` builtin moved the looping into the C code of the python interpreter. All in all, the functional approach reduced our execution time by an astonishing *97%*. Combined, the optimization of this section allowed us to execute data-generation (and aggregation) in mere hours instead of months. See Section IV-B for more specific numbers.

Optimization of query generation had an entirely different flavor. In particular, this component was not CPU constrained but *memory* constrained. Unfortunately, memory optimization was hindered by the lack of memory-profiling tools for large programs. Memory profiling of smaller, sample programs did yield many candidates for optimization. For example, switching from Python sets to lists yielded a 80% reduction in memory usage. Other optimizations included: reducing data-duplication in the aggregators, limiting the number of records an aggregator could collect, and (as mentioned above) postponing the call of the reduce method until after row generation was complete. Overall, the various optimizations decreased memory usage by a factor of fifty. However, the lack of large-scale profiling tools prevented us from reducing final memory consumption to the capability of our hardware. Even with 288GB of memory, we ultimately needed to break query-generation into sequential 'batches' in order to keep intermediate results within memory bounds.

Data Generator	Generation Rate
MUDD [5]	14 MB/s - 34 MB/s
DBGen [8]	9 MB/s
STSG	10 MB/s

Table I: A comparison of the other state of the art generators show that even though our data generator was written in Python, it generated data at a very similar rate to the other programs. All of these values have been normalized to the rate produced by a single process or core.

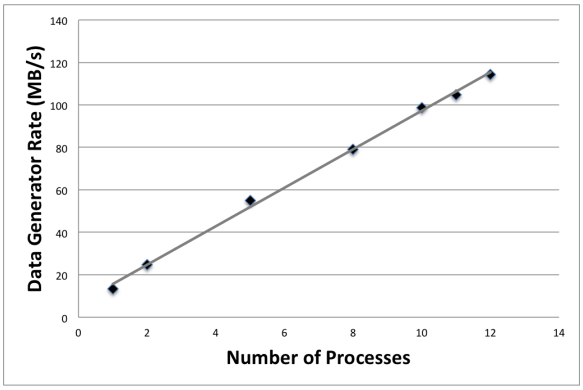

Figure 3: By changing the number of processes the scaling and parallelism of the generator was reflected in the data generation rate.

B. Performance Numbers

The performance of the test suite generator can be broken into two subsets, that of the data generator and that of the query generator.

Table I compares the per-core data-generation rate of SPAR test suite generator (STSG) to that of various other state-of-the-art generators.[5] Despite our use of Python, a high level programming language, our test suite generator performed comparable to other tools. Furthermore, and as Figure 3 shows, the performance of test suite generator scaled linearly with the number of processors. On the SPAR project, we used the generator on a Dell PowerEdge R710 server with 12 processors and were able to generate 10 TB in a little under 11 hours.

The performance of query generation was more difficult to characterize, but depended on three things: the number of rows generated, the number of queries generated, and whether those queries were simple or compound

[5]Note: the Flexible Data Generator is omitted due to lack of per-core data-rates.

(BOOL, THR or RANK). Recall that query generation was broken into three phases; query formulation, aggregation, and refinement. The first phase was negligible compared to the other two, and so we focus our discussion on the second two phases. Table II shows how the generation of EQ queries scales on the number of rows generated and the number of queries required. For both Table III and Table IV, a special reduced training set was used for expediency. Learning and query formulation took 36 seconds, combined, but this is not included in the data reported in those tables. In both tables, AGG represents the time to generate rows and aggregate ground truth results, REFINE is the time it took to refine all queries, and TOTAL is the overall time for a complete generation run.

As can be seen, the time required for aggregation scaled (roughly) linearly with the number of rows. The time required for refinement, on the other hand, scaled non-linearly. Some of our optimizations only applied at larger scales, making refinement a difficult operation to predict. However, experiment showed that all simple queries seem to share the same performance behavior, differing only by a constant factor. (These constant factors, normalized by that of EQ queries, are given in Table III.)

Compound queries, on the other hand, took considerably longer to generate than simple queries. Table IV shows the performance for BOOL queries. As can be seen, they took considerably longer than EQ queries and this primarily resulted from a more expensive refinement process. Although the search described in Section III is intelligent, it was still time-consuming. We put forth this area as the most promising location for future optimizations.

Queries \ Rows		10k	100k	1M
10	AGG	11	97	871
	REFINE	9	78	285
	TOTAL	**57**	**211**	**1192**
100	AGG	11	90	943
	REFINE	89	147	269
	TOTAL	**136**	**271**	**1248**
1000	AGG	14	112	940
	REFINE	343	678	587
	TOTAL	**391**	**826**	**1563**

Table II: The total time (in seconds) that EQ queries took to run on a Dell R710 with 12 processes given changing query numbers and number of rows.

Query Type	Total Runtime Overhead based on EQ
EQ	1.00
RNG	0.93
KWD	1.08
STM	0.94
WILD	1.00
SUB	1.01
XML	1.01

Table III: Under the same conditions as the EQ runs above, these were the factors by which the overall time would change if one were generating a different query type. Note that some query types were faster than EQ runs.

Queries \ Rows		10k	100k
10	AGG	13	105
	REFINE	20	84
	TOTAL	**69**	**225**
100	AGG	14	92
	REFINE	148	312
	TOTAL	**198**	**438**
1000	AGG	13	115
	REFINE	1170	2431
	TOTAL	**1219**	**2582**

Table IV: The total time (in seconds) that BOOL queries took to run on a Dell R710 with 12 processes given changing query numbers and number of rows.

ACKNOWLEDGMENT

The authors would like to acknowledge Oliver Dain for his aid in the design and implementation of the test suite generator.

REFERENCES

[1] "Python." [Online]. Available: www.python.org

[2] IARPA, "Broad agency announcement IARPA-BAA-11-01: Security and privacy assurance research (SPAR) program," February 2011.

[3] M. Varia, B. Price, N. Hwang, R. Cunningham, A. Hamlin, J. Herzog, J. Poland, M. Reschly, and S. Yakoubov, "Automated assessment of secure search systems," 2014, in submission.

[4] N. Bruno and S. Chaudhuri, "Flexible database generators." in *VLDB*, K. Bhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-. Larson, and B. C. Ooi, Eds. ACM, 2005, pp. 1097–1107. [Online]. Available: http://dblp.uni-trier.de/db/conf/vldb/vldb2005.html#BrunoC05

[5] J. M. Stephens and M. Poess, "Mudd: a multi-dimensional data generator," in *WOSP*, 2004, pp. 104–109.

[6] C. Mishra, N. Koudas, and C. Zuzarte, "Generating targeted queries for database testing," in *SIGMOD Conference*, 2008, pp. 499–510.

[7] D. R. Slutz, "Massive stochastic testing of sql," in *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, A. Gupta, O. Shmueli, and J. Widom, Eds. Morgan Kaufmann, 1998, pp. 618–622.

[8] "Transaction processing performance council." [Online]. Available: http://www.tpc.org/tpch/

[9] "Genealogy data: Frequently occurring surnames from census 2000." [Online]. Available: http://www.census.gov/genealogy/www/data/2000surnames/

[10] "Census 2000 5-percent public use microdata sample (pums) files." [Online]. Available: http://www2.census.gov/census_2000/datasets/PUMS/FivePercent/

[11] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[12] "Cython." [Online]. Available: cython.org