January 2022

Random Query Generator for Database Testing

Adam Dickinson

Everett Maus

Shannon Bales

Random Query Generator for Database Testing

ABSTRACT

An efficient test of a database is to subject it to random queries that are likely to succeed when evaluated on the database. Currently, automated query generation is difficult to use to target specific query shapes and difficult to adapt to diverse purposes. This disclosure describes a framework for generating random database queries by generating relational algebra trees and by building a query string from the algebra tree. The techniques enable the generation of useful, syntactically correct queries tuned towards particular depths, operator types, subtree structures, etc. Interesting algebra shapes can be targeted to test specific areas efficiently.

KEYWORDS

- Random query generator (RQG)
- Structured query language (SQL)
- Abstract syntax tree (AST)
- Builder graph
- Query engine
- Database test

BACKGROUND

An efficient test of a database is to subject it to random queries that are likely to succeed when evaluated on the database. A current technique to generate queries is to define what syntactically correct SQL looks like using a kind of grammar. The grammar can then be used to automatically generate query strings. While the technique is relatively easy to get started and can generate a wide variety of queries, there is a likelihood that the generated queries include invalid

ones. Also, this technique is difficult to use to target specific query shapes and to adapt to diverse testing purposes.

DESCRIPTION

This disclosure describes a framework for generating random database queries by generating relational algebra trees and building a query string from the algebra tree. Queries thus generated are syntactically correct. Interesting algebra shapes can be targeted to test specific areas efficiently.

The framework generates a random algebra (resolved abstract syntax tree, AST) and builds equivalent SQL statements from the algebra. The framework includes a two-pass procedure. A first pass builds a scan tree depth-first. In a second pass (on the way back up from leaf scans), expression trees are built. Each node in the tree is constructed by stateless builders which are assembled into a graph. All state is passed between builders in a context object which encodes constraints and identifiers which can be referenced. This maintains the independence of builders in the tree and limits the contract between builders. The described random query generator (RQG) uses catalog reflection to learn available types, functions, casts, tables, views, columns, etc., and is an example of an action generator in a system test.

```
// Get a sample catalog
Catalog* catalog = SampleCatalog::Get();

// Build a table list
const Table* keyvalue;
CHECK_OK(catalog->GetTable("KeyValue", &keyvalue));
vector<const Table*> tables;
tables.push_back(keyvalue);

// Create the strategy
RandomSelectionStrategy strategy; // Could provide seed for repros

// Create the builder graph
ResolvedNodeBuilderFactory builder_factory(tables, &strategy);

// Generate a query
std::unique_ptr<ResolvedStatement> query;
CHECK_OK(builder_factory.BuildQueryTree(&query));
LOG(INFO) << query->DebugString();
```

Fig. 1: Example code to generate a random query

Fig. 1 illustrates an example of C++ code that generates a random query targeting the table keyvalue in the catalog SampleCatalog, which is a pre-populated SQL catalog with predefined tables and data. In the example, a table list is provided to the query generator so that it can target the correct schema. The selection strategy is used to make choices where more than one option exists. Examples include choosing the kind of scan or expression to use as an input; choosing the size of a vector input; generating the value of a resolved literal; etc.

In the example code of Fig. 1, the random selection strategy is used. In this strategy, a random number generator is employed to make choices. This results in covering all of the options probabilistically over time, yielding reasonable test coverage over a long time period in relatively stress-focused test systems. The selection strategy abstracts decisions made by the

generator such that different strategies can be used, e.g., random selection, parallelized generation, etc.

The resolved node builder factory constructs, owns, initializes, and provides access to the builders in the builder graph. This factory includes all builders corresponding to resolved AST nodes. The BuildQueryTree() method uses the builder graph to generate and return a complete query tree. A parser turns SQL text into an AST, and a resolver resolves names and references in the AST into a resolved AST. A resolved AST is a tree made of resolved-node derived classes.

A builder class builds its own node and returns it to the builder above that needs that node as one of its inputs. The builder class also ensures that the tree below it is built (using other builders to build needed inputs). Each of the resolved-node derived classes is created by a builder class. Each of the inputs is created by another builder which is chosen by using a builder graph.

The builder graph is a directed cyclic graph used to describe the relationships between the different kinds of resolved nodes. This representation is convenient because the edges of the graph are not hard coded, enabling users to build a graph that is customized for their needs. Examples of customizations include adding and removing builders; dynamically reconfiguring the graph during generation; creating and adding new builders with targeted behavior; etc. Also, algorithms involved in generation are hidden from individual builders and instead live in shared code. The builders are almost entirely independent such that work on them can be split up and proceed in parallel. The nodes in the graph, the builders, are described above. The edges in the graph are represented by a class called builder relationship, which includes a pointer to the target builder as well as any properties that exist on the edge.

To represent abstract input types, the generator uses a delegating builder, which represents a choice among many potential builders. The delegating builder is initialized with a

set of other builders. All of the builders have a “BuildTree” command, providing a common interface. The “BuildTree” command of the delegating builder uses the selection strategy to choose one of the builders derived from the abstract type and delegates the build tree call to that builder.

Generating interesting random queries is fundamentally a constrained program synthesis problem. A series of abstract syntax tree nodes are built that are valid for a particular database schema and set of supported features. Some nodes (e.g., a function call to “+”) imply certain constraints on child nodes (e.g., both arguments to “+” must be integers). A number of options are tried that are likely to work for a particular node. If it is not possible to satisfy that constraint, e.g., if integers are not supported by the database, backtracking is performed, and a different function call (or expression) is chosen. Some global state, such as the parameters needed by the final query (and the constraints on them), are represented in a shared context object.

In this manner, the described techniques build interesting SQL queries given a schema, data, and a set of tree constraints. The techniques can produce runtime errors (e.g., divide by zero) for the purpose of testing and can reproduce environments or queries that fail.

CONCLUSION

This disclosure describes a framework for generating random database queries by generating relational algebra trees and building a query string from the algebra tree. The techniques enable the generation of useful, syntactically correct queries tuned towards particular depths, operator types, subtree structures, etc. Interesting algebra shapes can be targeted to test specific areas efficiently.