Automated SQL query generation for systematic testing of database engines

Conference Paper · January 2010

DOI: 10.1145/1858996.1859063 · Source: DBLP

Shadi Abdul Khalek
University of Texas at Austin

Automated SQL Query Generation for Systematic Testing of Database Engines

Shadi Abdul Khalek
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin TX, USA
shadi@mail.utexas.edu

Sarfraz Khurshid
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin TX, USA
khurshid@ece.utexas.edu

ABSTRACT

We present a novel approach for generating syntactically and semantically correct SQL queries for testing relational database systems. We leverage the SAT-based Alloy tool-set to reduce the problem of generating valid SQL queries into a SAT problem. Our approach translates SQL query constraints into Alloy models, which enable it to generate valid queries that cannot be automatically generated using conventional grammar-based generators.

Given a database schema, our new approach combined with our previous work on ADUSA, automatically generates (1) syntactically and semantically valid SQL queries for testing, (2) input data to populate test databases, and (3) expected result of executing the given query on the generated data.

Experimental results show that not only can we automatically generate valid queries which detect bugs in database engines, but also we are able to combine this work with our previous work on ADUSA to automatically generate input queries and tables as well as expected query execution outputs to enable automated testing of database engines.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools; H.2.3 [**Database Management**]: Languages—*SQL*

General Terms

Verification

Keywords

SQL, automated query generation, database management, Alloy.

1. INTRODUCTION

Software testing is the most commonly used methodology for validating the quality of software. However, testing is typically labor intensive and often amounts to more than one-half of the cost of software development. Testing applications that require complex inputs, such as database management systems (DBMSs) or compilers, is particularly expensive. Automation can significantly reduce the cost of testing as well as enable systematic testing, which can significantly increase the effectiveness of testing.

This paper presents a novel SAT-based approach to automate systematic testing of database management systems. There are three fundamental steps in testing a DBMS: (1) generating test queries with respect to a database schema, (2) generating a set of test databases (tables), and (3) generating oracles to verify the result of executing the queries on the input databases using the DBMS. Previous work has addressed each of these three steps but largely in isolation of the other steps [7, 8]. While a brute-force combination of existing approaches to automate DBMS testing is possible in principle, the resulting framework is unlikely to be practical: it will generate a prohibitively large number of test cases, which have a high percentage of tests that are redundant or invalid, and hence represent a significant amount of wasted effort. Some approaches, such as [6], target generating queries with cardinality constraints. Integrating query generators with data generators, however, is still either specialized [8], or sometimes not possible [6]. Several academic and commercial tools target the problem of test database generation [9, 10, 12]. Nevertheless, they do not support query generation nor test oracle generation. Recent work in query aware input generation [5] takes a parameterized SQL query as input and produces input tables and parameter values, but does not generate an oracle. Recent approaches introduced query-aware database generation [11, 13]. These approaches use the information from the queries as a basis to constrain the data generator to generate databases that provide interesting results upon query executions. Query-aware generation is gaining popularity in both DBMS and database application testing [14, 15] but requires queries to be provided manually.

A popular framework for query generation is the Random Query Generator (RQG) [4], which uses the SQL grammar as a basis of query generation - in the spirit of production grammars. Given a grammar RQG generates random queries and tests databases by running the tests against two or more databases and comparing their results. Since the query generation is purely grammar-based, it generates a large number of invalid queries as well as redundant ones. Moreover, validating that the queries generated are syntactically correct is hard and sometimes impossible to ensure [4].

The insight of our work is that a relational engine backed by SAT provides a sound and practical basis of a unified approach that supports all the three fundamental steps in DBMS testing and allows generation of a higher quality test suite: queries generated are valid, database states generated are query-aware, and expected outputs represent meaningful executions. Thus, each test case checks some core functionality of a DBMS.

```
CREATE TABLE students(        CREATE TABLE grades(
    id int,                       studetnID int,
    name varchar(50)              courseID int,
);                                grade int
                              );
```

Figure 1: Example database schema.

We make the following contributions:

- **Constraint-based query generation.** We present a framework that leverages the Alloy tool-set to model the language constraints of a useful subset of SQL and provides automated generation of valid queries (with respect to the constraints).
- **SAT for query generation.** We present a non-conventional use of SAT: to solve syntactic and semantic constraints of SQL language to enumerate queries as test inputs.
- **Case-studies.** We evaluate our approach using case-studies to demonstrate its usefulness in automation of DBMS testing.

2. EXAMPLE

In this section we give a simple example of automated SQL query generation which can not be generated using conventional grammar-based generators. We describe the input database schema and the corresponding SQL queries generated by our approach.

Let us consider a sample database schema as shown in Fig. 1. SQL statements such as primary and foreign keys constraints, as well as other statements which do not interfere with SQL query generation, are ignored in our approach since they are irrelevant to the problem we are solving. The SQL statements in Fig. 1 create two relations (also known as tables): (1) `students` table with two attributes, `id` of type `int`, and `name` of type `varchar`, (2) `grades` table with three attributes, `studentID` of type `int` representing a student ID number, `courseID` of type `int` representing the course ID number and `grade` of type `int` representing the grade which the student earned in that course.

Let us consider a subset of SQL grammar consisting of selecting up to two table attributes from either one or two tables cross joined. The terminal strings of the grammar are the table names and attribute names: *students, grades, id, name, sID,* and *courseID*. In addition, we consider that the grammar allows the use of aggregate functions when selecting a field. Let `MAX` and `MIN` be the only aggregate functions allowed. Below is the grammar of SQL queries that we will consider in this example:

```
QUERY ::= SELECT FROM
SELECT ::= 'SELECT' selectTerm+
FROM ::= 'FROM' (table | table JOIN table)
selectTerm ::= term | agg(term)
table ::= 'students' | 'grades'
term ::= 'id'|'name'|'studentID'|'courseID'|'grade'
agg ::= 'MAX' | 'MIN'
```

After automatically generating the complete Alloy model for this SQL grammar, the Alloy analyzer, based on SAT, will convert all Alloy formulas into boolean formulas and enumerates all possible solutions satisfying the model. We run the output through our concretization program to convert Alloy instances into complete SQL queries. For the grammar in this example, considering up to two SELECT terms, up to two FROM tables, and two aggregates, the Alloy Analyzer generates 186 unique non isomorphic[1] instances, which is the number of SQL queries expected. We then automatically translate each Alloy instance into a SQL query. Fig. 2 is a sample subset of the SQL queries generated by our approach.

[1]Two queries are considered isomorphic if they only differ in the order at which the SELECT terms, or the FROM tables are used.

```
SELECT courseID, studetnID FROM GRADES, STUDENT;
SELECT MAX (courseID), MAX (NAME) FROM GRADES, STUDENT;
SELECT MIN (courseID), MIN (NAME) FROM GRADES, STUDENT;
SELECT courseID FROM GRADES, STUDENT;
SELECT MAX (courseID), MIN (NAME) FROM GRADES, STUDENT;
SELECT MAX (NAME), MIN (courseID) FROM GRADES, STUDENT;
SELECT courseID, MIN (NAME) FROM GRADES, STUDENT;
SELECT courseID, MAX (NAME) FROM GRADES, STUDENT;
SELECT NAME, MAX (courseID) FROM GRADES, STUDENT;
SELECT NAME FROM STUDENT;
SELECT MIN (NAME) FROM STUDENT;
SELECT id FROM STUDENT;
SELECT MAX (NAME), MIN (id) FROM STUDENT;
SELECT MAX (id), MIN (NAME) FROM STUDENT;
SELECT id, MAX (NAME) FROM STUDENT;
...
```

Figure 2: SQL queries generated by our approach.

```
QUERY ::= SELECT FROM WHERE GROUP_BY HAVING
SELECT ::= 'SELECT' selectTerm+
selectTerm :: term | aggregate(term)
FROM ::= 'FROM' (table | table JOIN table)
WHERE ::= 'WHERE' term operator (term | value)
GROUP_BY ::= 'GROUP BY' term
HAVING ::= 'HAVING' term operator value
aggregate ::= 'MAX' | 'MIN' | 'AVG' | 'COUNT'
operator ::= '<' | '<=' | '>' | '>=' | '='
```

Figure 3: SQL Grammar supported

3. FRAMEWORK

In this section, we discuss the general algorithm of our approach. In our approach, we consider a subset of SQL query grammar; the complete grammar supported is shown in Fig. 3.

Alloy can be used to model a relational database schema[2]. To illustrate, consider the example used in section 2, the schema of the tables is shown in Fig. 1. Our approach generates an Alloy specification that represents both `students` and `grades` tables and each of their attributes. To systematically generate Alloy models for all tables, we model a general representation of tables and fields in Alloy as follows:

```
abstract sig FieldNames {}
abstract sig FieldTypes {}
abstract sig Field {
   name : one FieldNames,
   type : one FieldTypes
}
abstract sig TableNames {}
abstract sig Table {
   name : one TableNames,
   fields : some Field
}
```

We model aggregate functions by creating an abstract signature for all aggregates and extending this signature with the ones that we want to use. If we consider the aggregate functions `MIN` and `MAX` as in the grammar in our example, then the following Alloy code models these aggregates to be used in the query generation:

```
abstract sig AggregateNames {}
one sig MAX extends AggregateNames {}
one sig MIN extends AggregateNames {}
```

[2]Details about Alloy and Alloy Analyzer can be found in [2,3].

Modeling the SQL grammar in Alloy requires modeling each of the SELECT and FROM parts as separate entities. This guarantees the generation of syntactically correct queries. After modeling the query grammar in Alloy, we add constraints over the model which gives the ability to prune out queries which are either not useful or semantically incorrect. The following Alloy code models both the SELECT and FROM sections:

```
sig term {
   field : one Field,
   agg : lone AggregateNames
}
one sig SELECT {
   fields : some term
}
one sig FROM {
   tables: some Table
}
```

Our approach reads the database schema and automatically generates signatures and constraints for the tables and fields. First, we populate the `FieldNames` and `TableNames` with elements representing all the names of fields and tables that exist in the database schema. We use the `extends` keyword to extend any of the existing signatures in the model. We also use the multiplicity `lone` for these elements since they can be either singletons when used in a query or empty when not used. The following Alloy code covers all the field names and table names in our example that would be potentially used in queries generated:

```
lone sig id, name, studentID, courseID, grade
        extends FieldNames {}
one sig students, grades extends TableNames {}
```

We then automatically create a signature for each of the fields of the tables. These signatures extend the `Field` type. In addition, for each `Field` type extended we explicitly set the relation constraints, setting the name and type of every field explicitly. Similarly for each table in the schema, we create a signature extending the `Table` type.

This provides us with a backbone for syntactically SQL queries, we add to this model constraints to enforce semantically correct query generation. We mention some of these constraints here. To enforce that only attributes in the tables selected within the FROM clause can be chosen in the SELECT clause, we add the following Alloy fact to the model:

```
fact field_in_table {
   all f: term.field | some t: FROM.tables |
 f in t.fields
}
```

Similarly, to ensure that an attribute is selected only once in the SELECT clause, we add the following Alloy fact to the model:

```
fact unique_select_terms {
   all a, b : SELECT.fields.term |
  (a.field = b.field and a.agg = b.agg ) => a=b
}
```

4. CASE STUDIES

In this section we discuss the use of our framework in different case studies. We perform tasks for generating SQL queries based on different subsets of the SQL grammar.

Case#	SQL Grammar

1	QUERY ::= SELECT FROM
	SELECT ::= 'SELECT' selectTerm+
	FROM ::= 'FROM' (table | table JOIN table)

2	QUERY ::= SELECT FROM WHERE
	SELECT ::= 'SELECT' selectTerm+
	FROM ::= 'FROM' (table | table JOIN table)
	WHERE ::= 'WHERE' term operator (term | value)

3	QUERY ::= SELECT FROM GROUP_BY HAVING SELECT ::= 'SELECT' selectTerm+ FROM ::= 'FROM' (table | table JOIN table) GROUP_BY ::= 'GROUP BY' term HAVING ::= 'HAVING' term operator value
*	selectTerm ::= term | agg (term) term ::= 'id' | 'name' | 'studentID' | 'courseID' | 'grade' agg ::= 'MAX' | 'MIN' table ::= 'students' | 'grades'

Table 1: The SQL grammar used in each case study. (The * indicates common terminal values.)

4.1 Auto Query Generation

In each case study we use our approach to enumerate all possible valid queries for a given schema. We compare the approach by applying it to different subsets of the SQL grammar. We consider the same schema presented in Fig. 1 consisting of the two tables: `student` and `grades`. The subsets of SQL grammar that we consider are presented in Table 1. Case#1 consists of only SELECT and FROM clauses. For each test, we consider two cases: (1) up to one table in the FROM section, and (2) up to two tables in the FROM section. So, queries generated in (1) are inclusive to (2).

Table 2 shows the results of our approach. The total number of queries increases drastically for Case#2, this is because the fact that the WHERE clause can contain terms from the tables which are not constrained by the SELECT statement, and the fact that each term can be related to either another term or a value, thus the number of possible queries increases. Case#3, using up to one table in the FROM section, generates the minimum amount of queries. This is because both constraints for GROUP BY and HAVING must be satisfied in all the queries generated. In the grammar for Case#3, the GROUP BY and HAVING clauses are mandatory, thus limiting the output space.

4.2 Integration with ADUSA

One of the motivations for automatic generation of syntactically and semantically valid SQL queries is to automate the three fundamental steps in database testing. Our previous work on Automated Database Testing Using SAT (ADUSA) [1] uses model-based testing to perform (1) query-aware database generation to construct a useful test input suite covering various scenarios for query execution and (2) test oracle generation to verify query execution results on the generated databases. ADUSA takes as input (1) a database schema and (2) a SQL query. It populates the database with meaningful data and verifies the output of executing the query upon the database using an automatically generated oracle.

Our current approach, closes the gap of having a SQL query as a user-provided input for ADUSA. Given an input schema, our approach automatically generates valid SQL queries for testing. These queries along with the schema are used by ADUSA to perform a black-box testing on the database system. Having both approaches based on Alloy and SAT enables us combining the Alloy models to minimize number of variables used to solve the model. Tests generated using ADUSA were able to find and reproduce bugs in Oracle 11g, MySQl 4.0, and HSQLDB (injected bug) [1].

Case#	#Tables	Primary Vars	Total Vars	Clauses	Solving Time	Concret. Time/Query	#Queries
1	1	155	1586	2648	375	4.03	66
1	2	155	1586	2652	343	3.53	186
2	1	247	3013	5135	390	3.86	3456
2	2	247	3011	5129	422	5.13	27081
3	1	263	3209	5524	437	4.61	26
3	2	263	3210	5528	422	4.08	76

Table 2: Solving time of each of the case studies. The #Tables is the maximum number of tables in the FROM section. Primary variables and total variables are the Alloy variables used in generating the Boolean formula. The clauses are the Boolean clauses. Solving time is the SAT time to generate the first possible solution for the Boolean formula (next solutions take negligible time). Concretization time per query, is the processing time our approach does to concretize an Alloy instance into a SQL query in ms. The #Queries is the total number of queries generated for the specific case study.

5. FUTURE WORK AND CONCLUSION

Our approach shows how to use Alloy and the Alloy Analyzer to model a subset of SQL query grammar and its constraints to ensure the validity of the syntax and semantics of queries generated. The approach is extensible; we can systematically add support for a larger subset of SQL grammar. For example, we showed how to integrate in our framework the types for table attributes; these types can be used to add type checking constraints in the WHERE, GROUP BY, and HAVING clauses. SQL transactional grammar can be extended as well. DELETE statements can be introduced by modifying the grammar as: `DELETE FROM TABLE WHERE term in (SELECT term FROM table WHERE condition)`. The constraint that the term to be deleted is the same as the term to be selected is simple to write in Alloy. Nested SELECT statements can be extended by ensuring that the inner SELECT statements can have access to the outer SELECT terms but not vice-versa.

In conclusion, we presented a novel approach for SQL query generation to automate DBMS testing. Our approach automatically generates syntactically and semantically valid SQL queries. When combined with our previous work on ADUSA, we are capable of generating (1) test SQL queries, (2) query aware input databases, and (3) test oracles to verify the result of the query execution.

Our approach leverages the SAT-based Alloy tool-set. We systematically model the SQL queries in Alloy and add constraints to ensure semantic meaning of the queries, and then use the Alloy Analyzer to generate possible test queries out of the model.

We compared the output of our approach using different subsets of SQL grammar. Combined with our previous work, our framework allows finding new bugs and reproducing bugs in different database engines.

Acknowledgments

This material is based upon work partially supported by the NSF under Grant Nos. IIS-0438967, CCF-0845628, and CNS-0958231, and AFOSR grant FA9550-09-1-0351.

6. REFERENCES

[1] Shadi A. Khalek, Bassem Elkarablieh, Y. O. Laleye, and Sarfraz Khurshid. Query-Aware Test Generation Using a Relational Constraint Solver. In ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering , pages 238Ű-247, 2008.

[2] Daniel Jackson. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2), April 2002.

[3] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Tools and Algorithms for the Construction and Analysis of Systems, Vol. 4424, pages 632–647. 2007.

[4] MySQL Forge Random Query Generator. `http://forge.mysql.com/wiki/RandomQueryGenerator/`.

[5] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. Qex: Symbolic SQL Query Explorer Microsoft Research Technical Report MSR-TR-2009-2015, October 2009

[6] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating queries with cardinality constraints for DBMS testing. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1721–1725, 2006.

[7] Donald 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*, pages 618–622. Morgan Kaufmann, 1998.

[8] Meikel Poess and Jr. John M. Stephens. Generating thousand benchmark queries in seconds. In *VLDB'2004: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1045–1053, 2004.

[9] Nicolas Bruno and Surajit Chaudhuri. Flexible database generators. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1097–1107. VLDB Endowment, 2005.

[10] Kenneth Houkjaer, Kristian Torp, and Rico Wind. Simple and realistic data generation. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1243–1246. VLDB Endowment, 2006.

[11] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse query processing. In *ICDE*, pages 506–515. IEEE, 2007.

[12] IBM DB2. Test database generator. `www.ibm.com/software/data/db2imstools/db2tools/db2tdbg/`.

[13] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. Qagen: generating query-aware test databases. In *SIGMOD Conference*, pages 341–352, 2007.

[14] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 151–162, 2007.

[15] David Willmor and Suzanne M. Embury. An intensional approach to the specification of test cases for database applications. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 102–111, New York, NY, USA, 2006. ACM.