

CS6218. Principles of Programming Languages & Software Engineering

Week 2: Background



NUS
National University
of Singapore

National University of Singapore

Last Lecture

- ▶ Logistics
- ▶ Data-centric systems
 - ▶ What are they?
 - ▶ Why are they important?
 - ▶ Why is it difficult to maintain their correctness?
 - ▶ By what bugs can they be affected?

What's a Data-Centric System?

- ▶ Data is a central asset
 - ▶ **Intensional definition**
- ▶ Are AI applications included or not?
 - ▶ If yes, is any application that interacts with an environment data-centric?
- ▶ Let's settle on an **extensional definition**
 - ▶ Included: Database systems, data manipulation frameworks, data processing engines
 - ▶ Arguable: AI applications

Grading for Presentation

- ▶ Presentation grading
 - ▶ Understands the paper 30%
 - ▶ Own examples, Q/A, critical reflections, ...
 - ▶ Structure and content of presentation 30%
 - ▶ thread, well-motivated, figures, ...
 - ▶ Presentation style 20%
 - ▶ Appropriate speed, speaking freely, ...
 - ▶ Timing (25 - 30 minutes) 20%

Additional Notes About Presentations

- ▶ Read the papers thoroughly and critically
 - ▶ Sometimes requires reading parts of other papers
- ▶ I can offer to give feedback for the presentation
 - ▶ Coordinate with me when you want to send it to me

Grading for Project

- ▶ Initial report (30%)
 - ▶ Answered main questions
 - ▶ Structure and content
- ▶ Project artifact (20%)
 - ▶ Quality
 - ▶ Groups of two: contribution
- ▶ Project report (30%)
 - ▶ Structure and Content
- ▶ (Short) project presentation (20%)
- ▶ Bonus points for difficult projects, interesting ideas, ...

This Lecture

- ▶ A whirlwind tour on testing-related approaches
 - ▶ Ideas rather than details
 - ▶ Motivation for the upcoming paper presentations
- ▶ Test oracles for database systems

This Lecture

- ▶ A whirlwind tour on testing-related approaches
 - ▶ Ideas rather than details
 - ▶ Motivation for the upcoming paper presentations
- ▶ ~~Test oracles for database systems~~

Multiple students dropped the class,
allowing for a less tight schedule

Presentations

Presenters

Date	Presenter	Paper
30/08/22	Tan Yu Wei	Data-Oriented Differential Testing of Object-Relational Mapping
30/08/22	Nishita Dutta	APOLLO: automatic detection and diagnosis of performance regression
06/09/22	Wang Jianing	Metamorphic testing of Datalog engines
06/09/22	Jiang Yuancheng	Automatic Detection of Performance Bugs in Database Systems using
13/09/22	Zhang Anxing	SQUIRREL: Testing Database Management Systems with Language
13/09/22	Zhong Suyang	BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework
27/09/22	Rajdeep Singh Hundal	Search-based test data generation for SQL queries
27/09/22		
04/10/22		
04/10/22	Yang Ziyi	Synthesizing Analytical SQL Queries from Computation Demonstrations
11/10/22		
11/10/22	Kareem Shehata	SQLCheck: Automated Detection and Diagnosis of SQL Anti-Patterns
18/10/22		
18/10/22	Ho Han Kit Ivan	AIDA - Abstraction for Advanced In-Database Analytics

Presentations will start one week later (on 06/09/22) and subsequent presentations will be shifted and combined

(Manual) Testing

- ▶ Testing: Executing the program with the goal to find errors
- ▶ Dijkstra: “*program testing can be used very effectively to show the presence of bugs but never to show their absence*”
- ▶ Functional vs. non-functional requirements (e.g., performance)

Unit Testing

- ▶ Test a “unit” in isolation
- ▶ Often synonymous with **regression tests**
 - ▶ Check that previously-working functionality still works

Unit Testing

```
def bubble_sort(arr):  
    length = len(arr)  
    for i in range(length):  
        for j in range(0, length - i - 1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Unit Testing

```
def bubble_sort(arr):  
    length = len(arr)  
    for i in range(length):  
        for j in range(0, length - i - 1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[i]
```



Unit Testing

```
def bubble_sort(arr):  
    length = len(arr)  
    for i in range(length):  
        for j in range(0, length - i - 1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[i]
```



```
arr = [3, 1, 2]  
bubble_sort(arr)  
assert arr == [1, 2, 3] # AssertionError, actual: [1, 2, 1]
```

Unit Testing: MySQL

zlob_print.test

```
--source include/have_debug.inc
--source include/have_innodb_max_16k.inc

set global innodb_compression_level = 0;
create table t1 (f1 int primary key, f2 longblob)
  row_format=compressed, engine=innodb;
set debug='+d,innodb_zlob_print';
insert into t1 values (1, repeat('+', 1048576));
set debug='-d,innodb_zlob_print';
select f1, right(f2, 40) from t1;
drop table t1;
set global innodb_compression_level = default;
```

zlob_print.result

```
set global innodb_compression_level = 0;
create table t1 (f1 int primary key, f2 longblob)
row_format=compressed, engine=innodb;
set debug='+d,innodb_zlob_print';
insert into t1 values (1, repeat('+', 1048576));
set debug='-d,innodb_zlob_print';
select f1, right(f2, 40) from t1;
f1          right(f2, 40)
1
          ++++++
+
drop table t1;
set global innodb_compression_level = default;
```

https://github.com/mysql/mysql-server/blob/8.0/mysql-test/suite/innodb/t/zlob_print.test

https://github.com/mysql/mysql-server/blob/8.0/mysql-test/suite/innodb/r/zlob_print.result

(Manual) Testing

- ▶ How can we assess the quality of tests?
 - ▶ Number and importance of found bugs
 - ▶ Code coverage
 - ▶ Mutation testing

(Manual) Testing

- ▶ How can we assess the quality of tests?
 - ▶ Number and importance of found bugs
 - ▶ Code coverage
 - ▶ Mutation testing

Defining and evaluating
metric(s) is a potential project

(Manual) Testing

- ▶ How can we assess the quality of tests?
 - ▶ Number and importance of found bugs
 - ▶ Code coverage
 - ▶ Mutation testing

Code Coverage

- ▶ Idea: check which parts of the code are covered by tests
- ▶ Goals
 - ▶ Measure the quality of a test suite
 - ▶ Systematically test code
- ▶ Various granularities



Systematic Mistake Analysis of Digital Computer Programs

JOAN C. MILLER AND CLIFFORD J. MALONEY*
*U. S. Army Chemical Corps,
Fort Detrick, Frederick, Maryland*

Introduction

Effective program checkout is imperative to any complex computer program. One or more test cases are always run for a program before it is considered ready for application to the actual problem. Each test case checks that portion of the program actually used in its computation. Too often, however, mistakes show up as late as several months (or even years) after a program has been put into operation. This is an indication that portions of the program called upon only by rarely occurring input conditions have not been properly tested during the checkout stage.

In order to rely with confidence upon any particular program, it is not sufficient to know that the program works most of the time or even that it has never made a mistake so far. The real question is whether it can be counted upon to fulfill its functional specifications successfully every single time. This means that, after a program has passed the checkout stage, there should be no possibility that an unusual combination of input data or conditions may bring to light an unexpected mistake in the program. Every portion of the program must be utilized during checkout in order that its correctness may be confirmed.

The purpose of work reported here has been to develop a systematic way in which a programmer may test all realistic combinations of input data, and hence all portions of a given program. Although at first this seems to be an arduous task, its handling is simplified by an orderly approach, and its reward is a greater assurance of the correctness and reliability of the program. The *potential* number of test cases is given by 2^B , where B is the number of branchpoints in the flowchart. The computer could be programmed to generate all of these cases automatically. However, if the number of branchpoints is at all large,

* Now at the Division of Biological Standards, National Insti-

the total number of potential cases would become impractically large, though most of these would be unrealistic or identical to other cases and hence unnecessary to test. The analysis set forth in this paper has as its object the determination by straightforward means of the full list of possible—as opposed to potential—test cases, their method of convenient generation and their use in the location of specific mistakes in the program.

The approach taken here of using varied input conditions as the means to insure thorough testing of the program is particularly applicable to business problems—because a wide variety of input is one of the major aspects of such a problem. The method can be extended, however, to deal effectively with scientific problems, even though some of their branchpoints will not be dependent upon the input data. The procedure should therefore be useful as a method for systematic mistake analysis of computer programs irrespective of the type of program to be analyzed.

Background

Probably the most common method of mistake detection is the running of a test case whose answers have already been calculated elsewhere for comparison. Carefully chosen test data can be quite successful in pointing up the major mistakes. Other methods of mistake prevention which can be effective are (a) to code into the program certain checking devices such as intermediate check sums, and (b) to write the code in such a way that it will be easy to check, even at the expense of certain sophistications and time-saving devices [1]. It is always helpful to make a detailed manual check of the code as written, prior to the first machine run. In addition to reviewing the coding, a second flowchart can be drawn, this time prepared directly from the coding, to determine whether the program executes its instructions according to the original plan.

An increasing number of checkout methods make use of the facilities of the machine itself. Some of the more common methods [2] are manual step-by-step operation, dumping of memory, a tracing program with automatic skips and the use of the break-jump switch, which will stop the program and jump to any specified location. Jacoby and Layton [3] reported on an automated diagnostic program that runs the program to be tested, creating a trace record for later analysis by the diagnostic itself. Halbit [4] has described a program that enables the com-

<https://dl.acm.org/doi/pdf/10.1145/366246.366248>

Code Coverage: Common Metrics

- ▶ Function coverage = $\frac{\# \text{ executed functions}}{\# \text{ functions}} \times 100$
- ▶ Statement coverage = $\frac{\# \text{ executed statements}}{\# \text{ statements}} \times 100$
- ▶ Branch coverage = $\frac{\# \text{ executed branches}}{\# \text{ branches}} \times 100$

Code Coverage: Function Coverage

```
def bubble_sort(arr):  
    length = len(arr)  
    for i in range(length):  
        for j in range(0, length - i - 1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[i]
```

- ▶ Goal: reach 100% function coverage

Code Coverage: Function Coverage

```
def bubble_sort(arr):  
    length = len(arr)  
    for i in range(length):  
        for j in range(0, length - i - 1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[i]
```

```
bubble_sort([])
```

Code Coverage: Statement Coverage

```
def bubble_sort(arr):  
    length = len(arr)  
    for i in range(length):  
        for j in range(0, length - i - 1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[i]
```

```
bubble_sort([2, 1])
```

Code Coverage: Branch Coverage

```
def bubble_sort(arr):  
    length = len(arr)  
    for i in range(length):  
        for j in range(0, length - i - 1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[i]
```

```
bubble_sort([3, 2, 1])
```


Code Coverage

- ▶ Low coverage means that the code is not tested
- ▶ 100% coverage does not imply that the program is fully tested
- ▶ Often reported in the evaluation of testing approaches
- ▶ SQL-specific coverage criteria have been proposed

Full predicate coverage for testing SQL database queries

Javier Tuya *, M^a José Suárez-Cabal and Claudio de la Riva
Department of Computer Science, University of Oviedo,
Campus of Viesques, s/n, 33204 Gijón (SPAIN)

SUMMARY

In the field of database applications a considerable part of the business logic is implemented using a semi-declarative language: the Structured Query Language (SQL). Because of the different semantics of SQL compared to other procedural languages, the conventional coverage criteria for testing are not directly applicable. This paper presents a criterion specifically tailored for SQL queries (SQLFpc). It is based on Masking Modified Condition Decision Coverage (MCDC) or Full Predicate Coverage and takes into account a wide range of the syntax and semantics of SQL, including selection, joining, grouping, aggregations, subqueries, case expressions and null values. The criterion assesses the coverage of the test data in relation to the query that is executed and it is expressed as a set of rules that are automatically generated and efficiently evaluated against a test database. The use of the criterion is illustrated in a case study which includes complex queries.

KEY WORDS: software testing; database testing, MCDC, Full Predicate Coverage, SQL

* Correspondence to: Javier Tuya, Departamento de Informática, Universidad de Oviedo, Campus de Viesques s/n, E-33207 Gijón (SPAIN)
Tel.: (34) 985 182 049, FAX: (34) 985 181 986
E-mail: tuyaj@uniovi.es

Contract/grant sponsor: Department of Science and Innovation (Spain) and ERDF Funds; contract/grant number: TIN2007-67843-C06-01

Contract/grant sponsor: Government of the Principality of Asturias; contract/grant number: CN-07-168

Contract/grant sponsor: Government of Castilla-La Mancha; contract/grant number: PAC08-121-1374

1. INTRODUCTION

Database applications involve the management of large amounts of data stored and organized in many tables. Although there have been developments in object oriented databases and more recently in eXtensible Markup Language (XML) databases, most applications still maintain the data using Relational Database Management Systems (DBMS) that provide a high performance and a high degree of scalability and dependability. Different solutions to manage the data have been developed (such as persistence systems or object/relational mappings). However, the Structured Query language (SQL) [1] is still widely used, especially when its full expressive

Black-, Grey-, and White-box Approaches

- ▶ No standardized definition
- ▶ **Black-box**: no internal information
 - ▶ For example, based on a function's documentation
- ▶ **White-box**: internal information
 - ▶ For example, based on a function's implementation
- ▶ **Grey-box**: some internal information (e.g., coverage)
 - ▶ Relevant for Fuzzing

(Manual) Testing

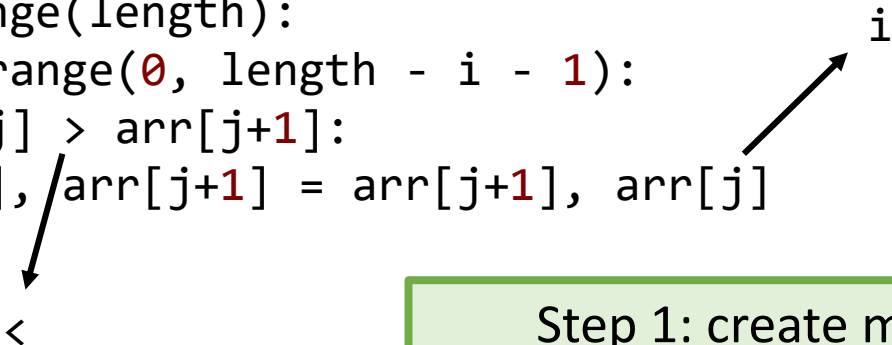
- ▶ How can we assess the quality of tests?
 - ▶ Number and importance of found bugs
 - ▶ Code coverage
 - ▶ Mutation testing

Mutation Testing

- ▶ Goal
 - ▶ Evaluate the quality of existing tests
 - ▶ Derive new tests
- ▶ Idea: mutate code in the program, assuming that a test case “kills” the mutant
- ▶ Percentage of mutants killed → quality of the test suite
- ▶ White-box testing technique

Unit Testing

```
def bubble_sort(arr):  
    length = len(arr)  
    for i in range(length):  
        for j in range(0, length - i - 1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[j]
```



Step 1: create mutants

Unit Testing

```
def bubble_sort(arr):  
    length = len(arr)  
    for i in range(length):  
        for j in range(0, length - i - 1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[i]
```

run_tests()

Step 2: run test suite for
every mutant: mutants is
killed if test cases fails

Unit Testing

```
def bubble_sort(arr):  
    length = len(arr)  
    for i in range(length):  
        for j in range(0, length - i - 1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[i]
```

$$\text{Mutation score} = \frac{\# \text{ killed mutants}}{\# \text{ mutants}} \times 100$$

Step 3: Compute a **mutation score** based on the percentage of killed mutants

Automated Testing & Fuzzing

- ▶ Goal: automatically generate tests or inputs to find bugs
- ▶ Two related perspectives
 - ▶ Automated testing
 - ▶ Fuzzing

Automated Testing & Fuzzing



CISPA
HELMHOLTZ CENTER FOR
INFORMATION SECURITY

Fuzzing: A Tale of Cultures

Andreas Zeller

Fuzzing Workshop @ NDSS'22 • April 24, 2022

Automated Testing & Fuzzing

Automated Testing

- ▶ Software engineering community
- ▶ Test own programs
- ▶ Knowledge about the domain & applications
- ▶ Test oracles

Fuzzing

- ▶ Security community
- ▶ Test other programs
- ▶ Minimal assumptions
- ▶ Security vulnerabilities

Automated Testing & Fuzzing

Automated Testing

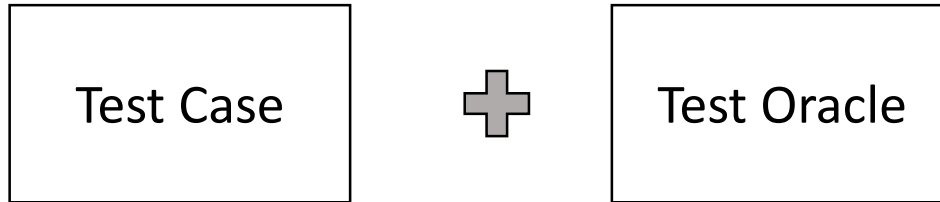
- ▶ Software engineering community
- ▶ Test own programs
- ▶ Knowledge about the domain & applications
- ▶ Test oracles

Fuzzing

- ▶ Security community
- ▶ Test other programs
- ▶ Minimal assumptions
- ▶ Security vulnerabilities

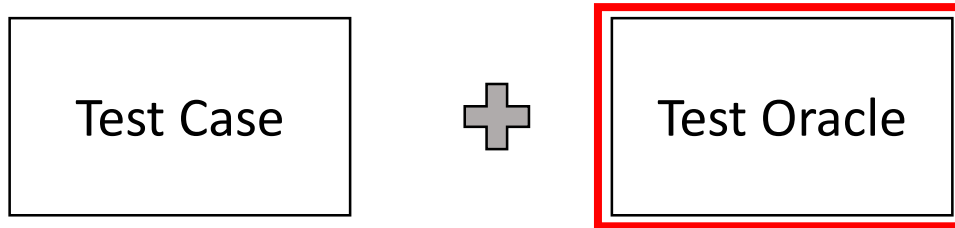
Automated Testing

- ▶ Automation of test case generation and the test oracle



Automated Testing

- ▶ Automation of test case generation and the test oracle



Test Oracle

Incorrect result!



“a test oracle (or just oracle) is a mechanism for determining whether a test has passed or failed”

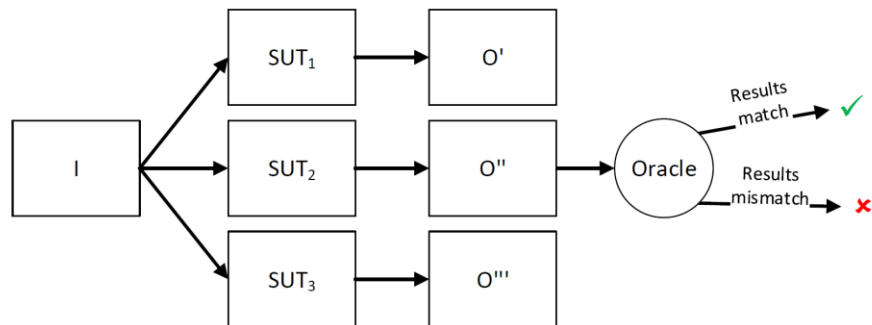
Test Oracle

- ▶ (Executable) program specification unavailable
- ▶ Complete vs partial specification

Differential Testing

- ▶ Systems compared
 - ▶ Different implementations (e.g., JVMs)
 - ▶ Versions (e.g., an old and new version)
 - ▶ Configurations (e.g., `-O0` vs `-O3`)
- ▶ Goals: correctness, performance, ...

Differential Testing



Black-box or white-box
technique?

Differential Testing

```
sorting_algorithms = [bubble_sort, merge_sort, insertion_sort]
while True:
    arr = get_random_array() # e.g., [3, 1, 2]
    sorted_arrays = [alg(arr) for alg in sorting_algorithms]
    all_same = all(sorted_arr == sorted_arrays[0] for sorted_arr in
sorted_arrays)
    assert all_same, sorted_arrays
```

Slutz, VLDB 1998

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

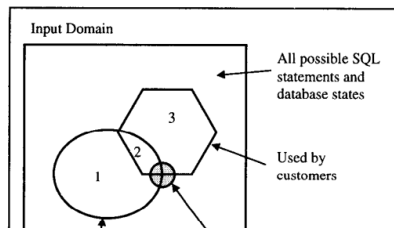
Good test coverage for commercial SQL database systems is very hard. The *input domain*, all SQL statements, from any number of users, combined with all states of the database, is gigantic. It is also difficult to verify output for positive tests because the semantics of SQL are complicated.

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 proc-

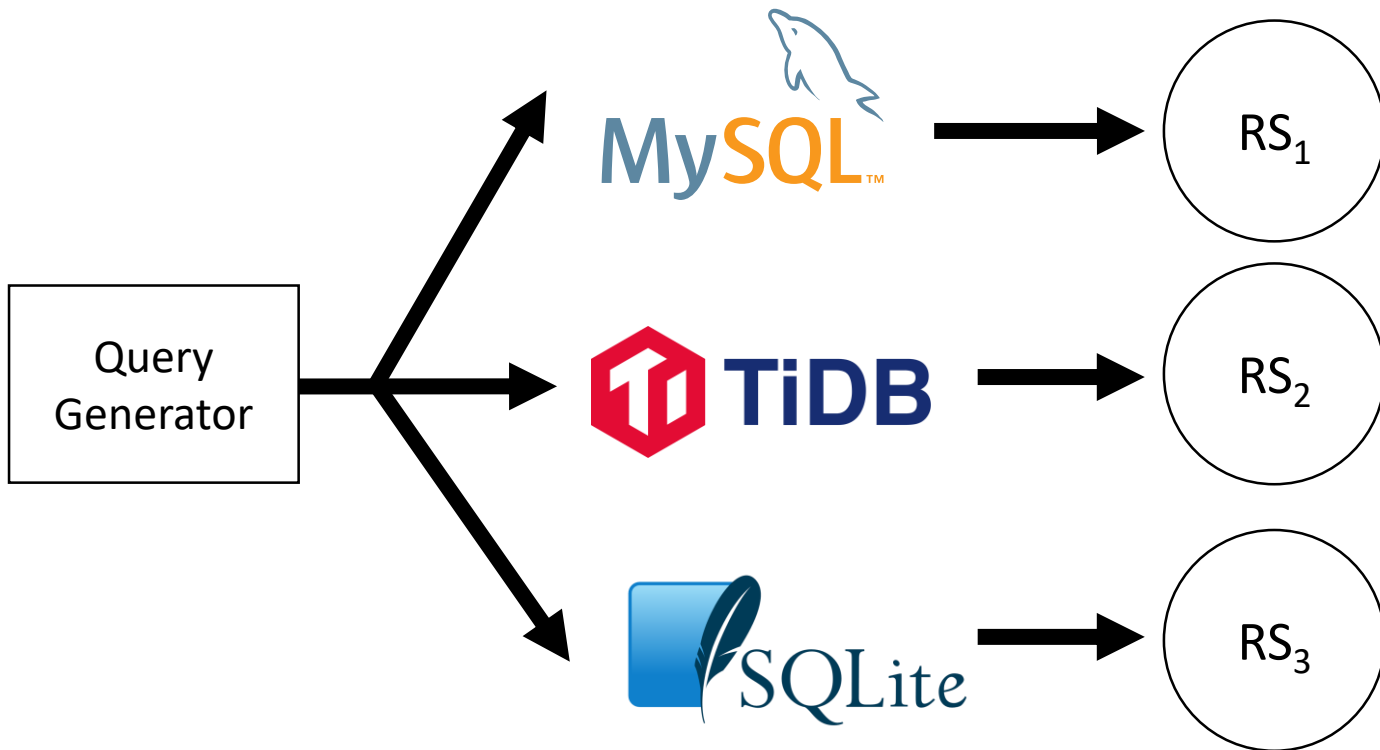
istribute the SQL statements in useful regions of the input domain. If the distribution is adequate, stochastic testing has the advantage that the quality of the tests improves as the test size increases [TFW93].

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.

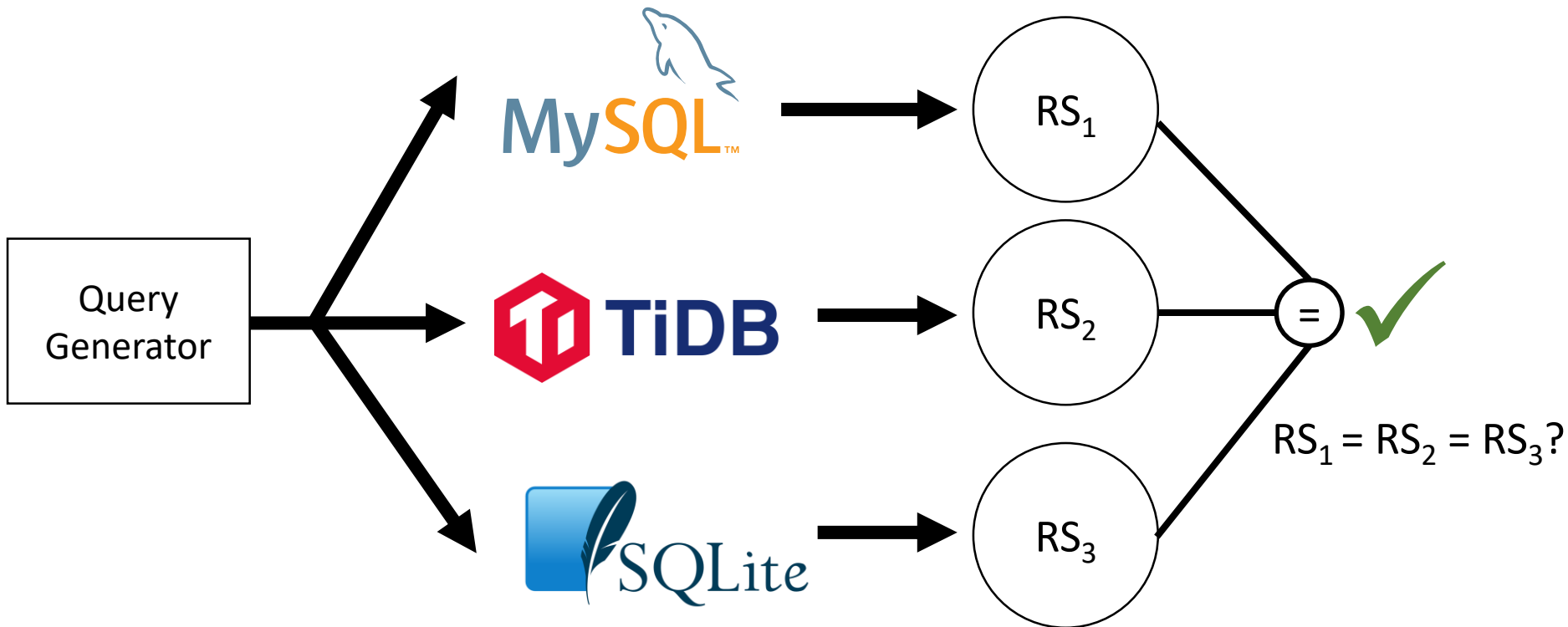


Slutz, VLDB 1998

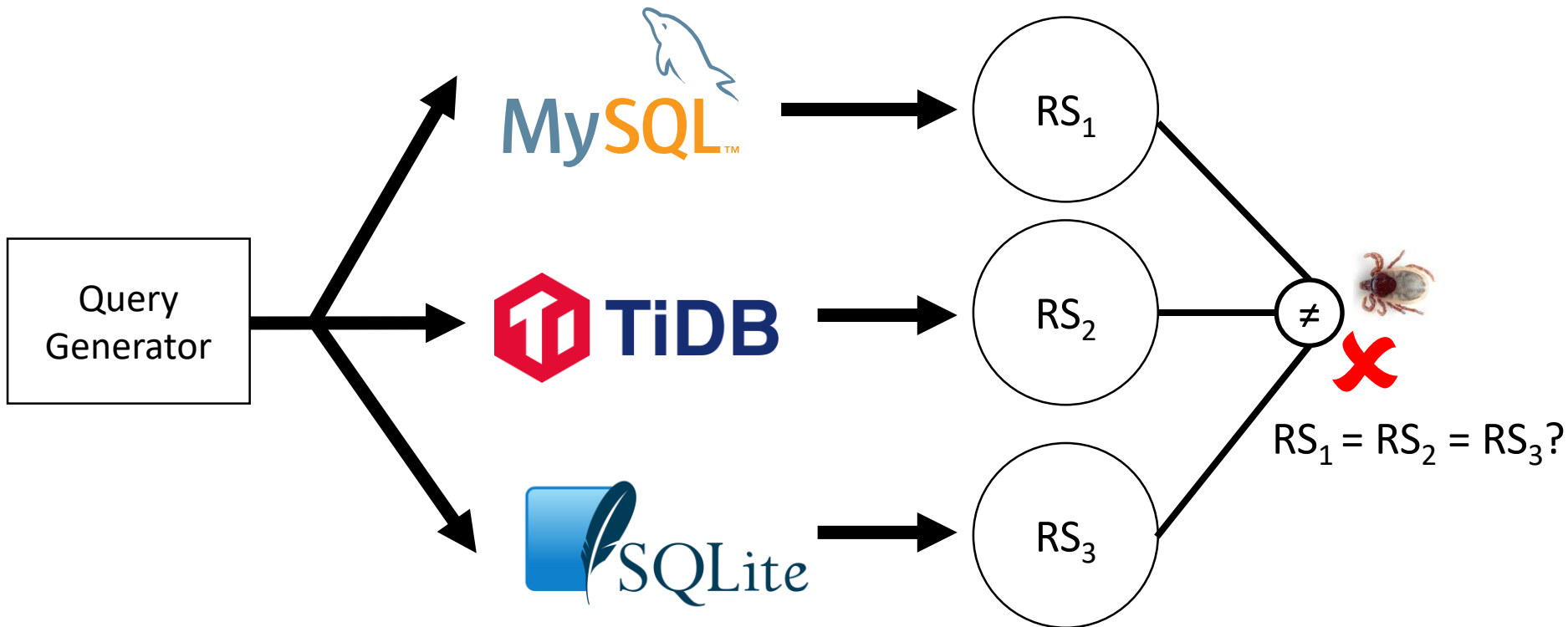


$RS_1 = RS_2 = RS_3?$

Slutz, VLDB 1998



Slutz, VLDB 1998



Ghit et al., DBTest '20

Useful and simple technique

SparkFuzz: Searching Correctness Regressions in Modern Query Engines

Bogdan Ghit
bogdan.ghit@databricks.com
Databricks Inc.

Nicolas Poggi
nicolas.poggi@databricks.com
Databricks Inc.

Josh Rosen
joshrosen@databricks.com
Databricks Inc.

Reynold Xin
rxin@databricks.com
Databricks Inc.

Peter Boncz
peter.boncz@cw.nl
Centrum Wiskunde & Informatica

ABSTRACT

With more than 1200 contributors, Apache Spark is one of the most actively developed open source projects. At this scale and pace of development, mistakes are bound to happen. In this paper we present SparkFuzz, a toolkit we developed at Databricks for uncovering correctness errors in the Spark SQL engine. To guard the system against correctness errors, SparkFuzz takes a fuzzing approach to testing by generating random data and queries. SparkFuzz executes the generated queries on a reference database system such as PostgreSQL which is then used as a test oracle to verify the results returned by Spark SQL. We explain the approach we take to data and query generation and we analyze the coverage of SparkFuzz. We show that SparkFuzz achieves its current maximum coverage relatively fast by generating a small number of queries.

ACM Reference Format:

Bogdan Ghit, Nicolas Poggi, Josh Rosen, Reynold Xin, and Peter Boncz. 2020. SparkFuzz: Searching Correctness Regressions in Modern Query Engines. In *Workshop on Testing Database Systems (DBTest '20)*, June 19, 2020, Portland, OR, USA. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3395032.3395327>

1 INTRODUCTION

Early data analytics frameworks such as MapReduce enabled users to simplify the execution of their big data workloads by means of a powerful, but low-level procedural programming interface. To cope with this limitation, systems such as Hive [24], Impala [19], and Spark SQL [13] expose relational interfaces to big data applications, thus providing richer automatic optimizations. As a result, the design of mechanisms for improving the performance of data analytics systems is an active research area both in academia and industry [15, 16, 25]. With an increasingly complex architecture, such systems are difficult to test with good coverage in practice. Developers are at risk to incorporate bugs, which may not only negatively impact the system performance, but may also alter the correctness of the results. In this paper we present the design and

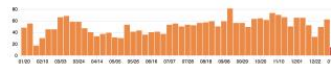


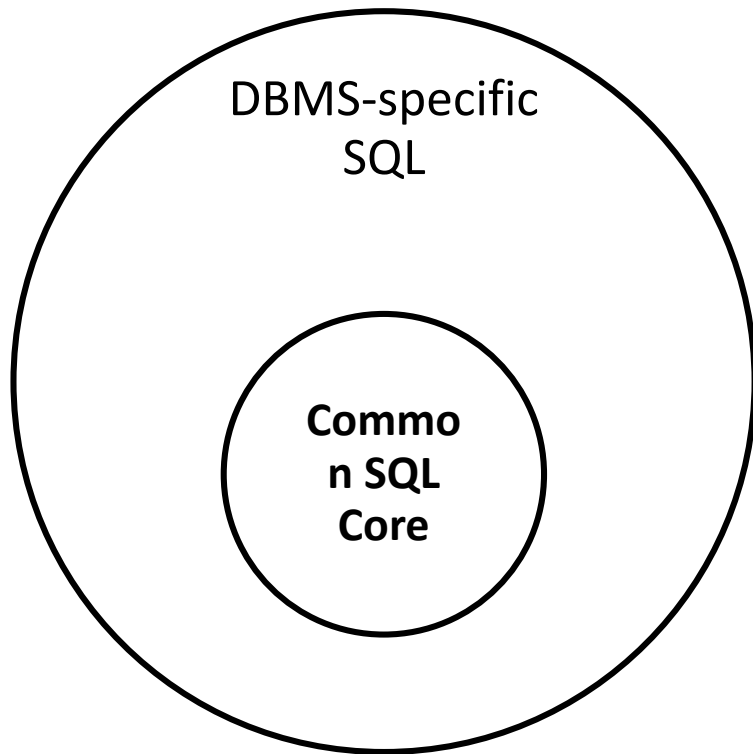
Figure 1: The distribution of code contributions in the Apache Spark open source project in the trailing year.

implementation of SparkFuzz, a toolkit for automatically generating SQL test cases which consist of random data and queries.

With powerful processing features and simple programming interface catalyzing its wide adoption, Spark has recently become the de facto framework for big data analytics [21]. Figure 1 shows that the Spark open source code base changes at a pace of tens of commits per day and so, mistakes are bound to happen. To guard the framework against errors, developers add unit tests which often results in a significant engineering effort. Spark has roughly the same amount of source and testing code. The effectiveness of such tests is however relatively low because they are restricted to specific operations on fixed inputs which cannot cover all possible code paths. Furthermore, data analytics frameworks are also prone to relatively high variability when the input dataset changes [17]. Therefore, standard testing techniques fail to capture data-dependent runtime interactions in these frameworks.

Catalyst, the Spark query optimizer, employs pattern-matching to express composable rules in a Turing-complete language, while offering a general framework for transforming trees. Catalyst modifies the user queries through tree transformations which are called rules. Such rules are grouped into multiple batches which are executed until the query plan reaches a fixed point – the tree stops changing after applying the same set of rules. Combining the supported set of rules in different ways typically diversifies the generated code paths and uncover regressions that may remain hidden otherwise. Testing all possible combinations of rules is however a

Slutz, VLDB 1998



“[...] proved to be extremely useful, but only for the **small set of common SQL**”

Differences Between SQL Dialects

Y Hacker News new | past | comments | ask | show | jobs | submit

1. ▲ Risk Everything (archive.org)
99 points by sutro 2 hours ago | hide | 17 comments
2. ▲ An odd discovery on Spotify (robinsloan.com)
390 points by breatheknow 6 hours ago | hide | 126 comments
3. ▲ Dell XPS 13 Plus developer edition: Now available with Ubuntu 22.04 LTS (dell.com)
48 points by nixcraft 1 hour ago | hide | 40 comments
4. ▲ Republishing a fork of the sanctioned Tornado Cash repositories (twitter.com/matthew_d_green)
256 points by Andrew_nenakhov 8 hours ago | hide | 197 comments
5. ▲ Google refuses to reinstate account after man took medical images of son's groin (theguardian.com)
224 points by sandebert 2 hours ago | hide | 108 comments
6. ▲ Parsing SQL (tomassetti.me)
50 points by teleforce 4 hours ago | hide | 14 comments

What are the limits of SQL? After a while you learn of a couple of issues:

- **there is not one SQL**, but many variations of it. The SQLs implemented in SQLite, MySQL, PostgreSQL, etc. are all a bit different

<https://tomassetti.me/parsing-sql/>

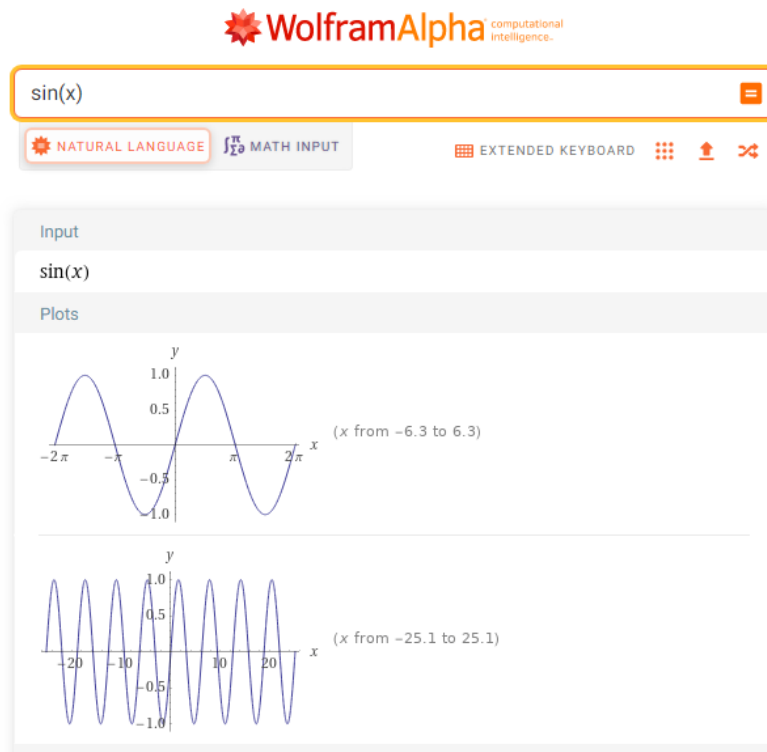
Differential Testing

- ▶ “Obvious” approach for systems that implement the same semantics
- ▶ Key challenge: how to generate valid test cases that are indeed expected to produce the same result
 - ▶ C compilers: avoid undefined behavior

Project idea: apply differential testing to a new domain or test for a different requirement (e.g., performance)

Metamorphic Testing

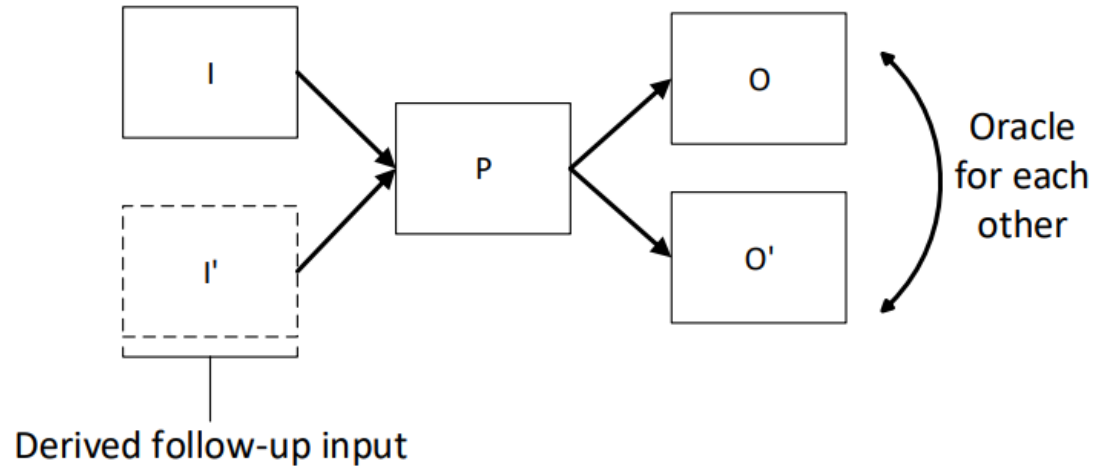
- ▶ Testing “untestable” systems
 - ▶ no ground truth
- ▶ Generate a follow-up test cases
- ▶ Black-box approach
- ▶ Requires a *metamorphic relation*
 - ▶ Prime example: $\sin(\pi - x) = \sin x$



<https://www.wolframalpha.com/input?i=sin%28x%29>

Metamorphic Testing

Metamorphic Testing



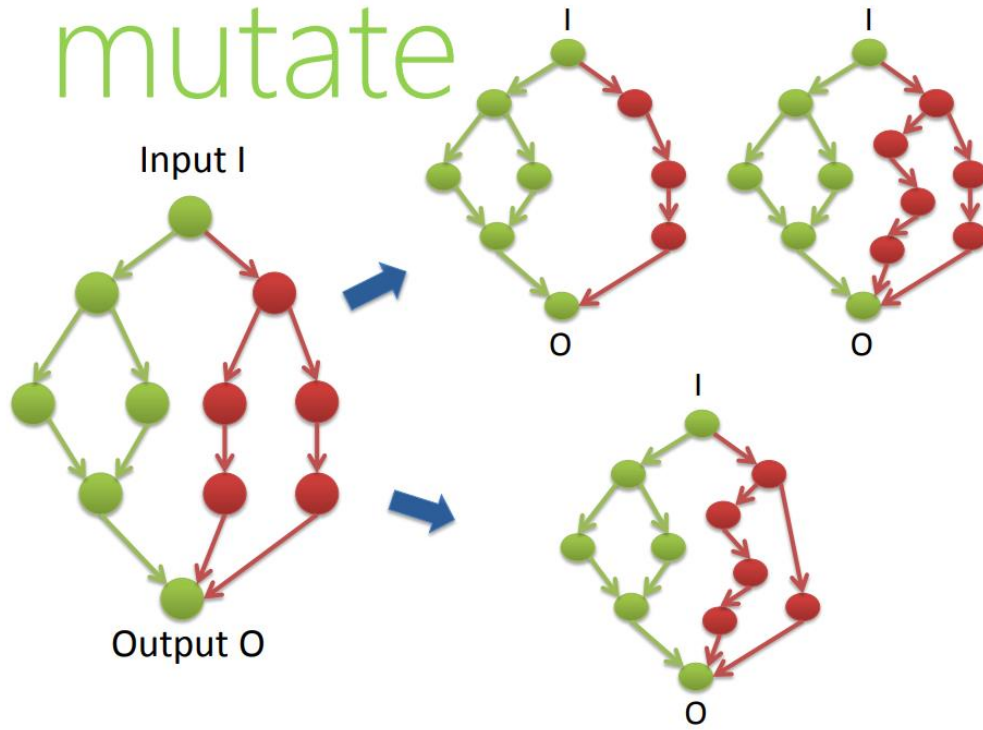
Metamorphic Testing

```
while True:
    arr = get_random_array()
    if len(arr) >= 1:
        sorted_arr = bubble_sort(arr)
        random_elem = random.choice(sorted_arr)
        arr.remove(random_elem)
        sorted_smaller_arr = bubble_sort(arr)
        sorted_arr.remove(random_elem)
        assert sorted_arr == sorted_smaller_arr,
str(sorted_arr) + str(sorted_smaller_arr)
```

Check whether **the relative order of sorted elements is maintained** when an element is removed from an input array

Example: EMI

mutate



Xinyue et al. (DBTest'20)

Testing Query Execution Engines with Mutations

Xinyue Chen
chenxy20@cs.washington.edu
University of Washington

Chenglong Wang
clwang@cs.washington.edu
University of Washington

Alvin Cheung
akcheung@cs.berkeley.edu
University of California, Berkeley

ABSTRACT

Query optimizer engine plays an important role in modern database systems. However, due to the complex nature of query optimizers, validating the correctness of a query execution engine is inherently challenging. In particular, the high cost of testing query execution engines often prevents developers from making fast iteration during the development process, which can increase the development cycle or lead to production-level bugs. To address this challenge, we propose a tool, MUTASQL, that can quickly discover correctness bugs in SQL execution engines. MUTASQL generates test cases by mutating a query Q over database D into a query Q' that should evaluate to the same result as Q on D . MUTASQL then checks the execution results of Q' and Q on the tested engine. We evaluated MUTASQL on previous SQLite versions with known bugs as well as the newest SQLite release. The result shows that MUTASQL can effectively reproduce 34 bugs in previous versions and discover a new bug in the current SQLite release.

In this paper, we describe MUTASQL, a new light-weight mutation testing engine can both *efficiently discover* and *effectively report* SQL engine bugs that can be used during development. Motivated by mutation testing in software engineering literature [2], MUTASQL achieves the “best of both worlds” between hand-written and randomly generated test cases by allowing developers to provide light-weight *seed queries* and optional *rewrite rules*. Based on the provided seed queries and rules, MUTASQL will intelligently generate test cases such that they are semantically equivalent to the seeds, making it easy to validate the results returned by the query engine under test.

Concretely, MUTASQL consists of two components: (1) a *mutation engine* that changes queries iteratively starting from seed to more complex ones in searching for engine bugs, and (2) a *query simplifier* that simplifies and generalizes a reported query that triggers bugs, with the goal to derive a minimal, human-readable query for developers that can be used for debugging.

Xinyue et al. (DBTest'20)

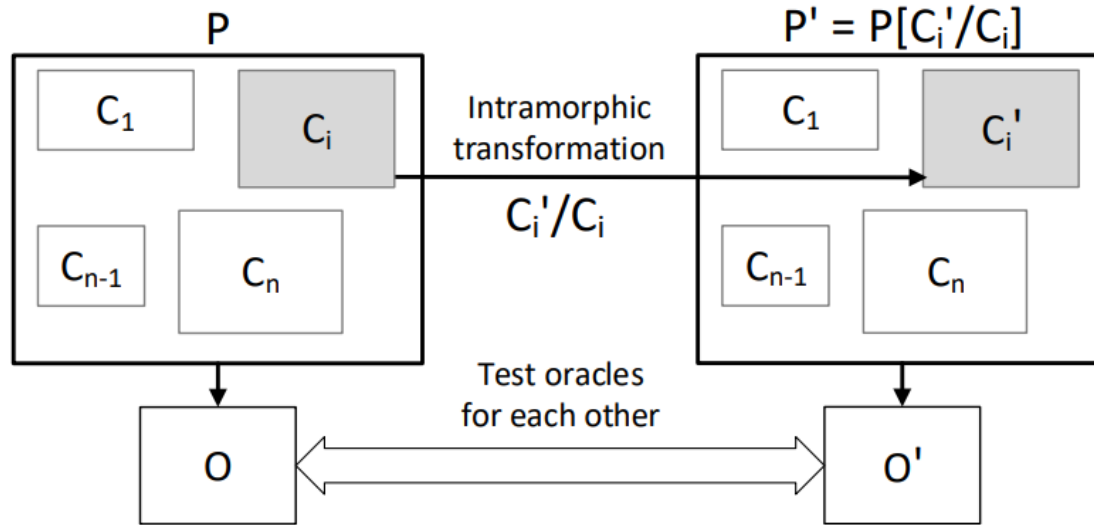
$$(Q = \begin{array}{l} \text{Select } c1, \dots \\ \text{From } t \\ \text{Where } p \end{array}, D) \xrightarrow[\text{no duplicate tuples}]{\text{if } q(D) \text{ contains}} Q' = \begin{array}{l} \text{Select } c1, \dots \\ \text{From } t \\ \text{Where } p \\ \text{Group By } c1, \dots \end{array}$$

Manually-specified rules that, when applied, guarantee that the result does not change

Intramorphic Testing

- ▶ Idea: rather than changing an input or test case, change the program under test
- ▶ White-box testing approach

Intramorphic Testing



Intramorphic Testing

```
while True:
    arr = get_random_array()
    sorted_arr = bubble_sort(arr.copy())
    reverse_sorted_arr = bubble_sort_reverse(arr.copy())
    assert sorted_arr.reverse() == reverse_sorted_arr,
    str(sorted_arr) + ' ' + str(reverse_sorted_arr)
```

Property-based Testing

- ▶ Idea: specify properties that hold for a function (or component)
 - ▶ (Randomly) generate data to test whether these properties indeed hold
- ▶ Pioneered in QuickCheck
 - ▶ Implemented in ~300 lines of code

Property-based Testing

```
def property_idempotent(l):  
    double_rev_list = l.copy()  
    bubble_sort(l)  
    bubble_sort(double_rev_list)  
    bubble_sort(double_rev_list)  
    return l == double_rev_list
```

Property-based: Expected Errors

```
private SQLQueryAdapter generate() {  
    sb.append("INSERT INTO ");  
    DuckDBTable table = globalState.getSchema().getRandomTable(t -> !t.isView());  
    List<DuckDBColumn> columns = table.getRandomNonEmptyColumnSubset();  
    sb.append(table.getName());  
    sb.append("(");  
    sb.append(columns.stream().map(c -> c.getName()).collect(Collectors.joining(", ")));  
    sb.append(")");  
    sb.append(" VALUES ");  
    insertColumns(columns);  
    DuckDBErrors.addInsertErrors(errors);  
    return new SQLQueryAdapter(sb.toString(), errors);  
}
```

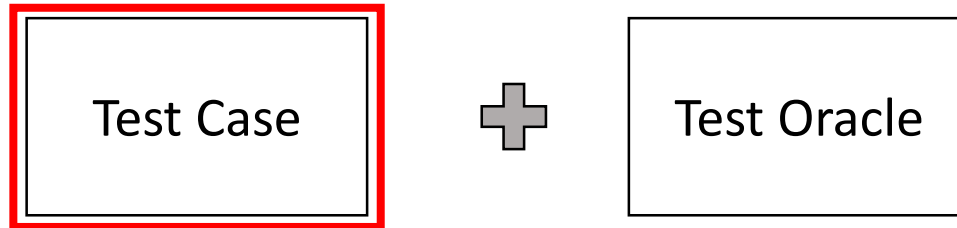
Property-based: Expected Errors

```
public static void addInsertErrors(ExpectedErrors errors) {  
    errors.add("NOT NULL constraint failed");  
    errors.add("PRIMARY KEY or UNIQUE constraint violated");  
    errors.add("duplicate key value violates primary key or unique constraint");  
    errors.add("can't be cast because the value is out of range for the destination type");  
    errors.add("Could not convert string");  
    errors.add("timestamp field value out of range");  
    errors.add("Unimplemented type for cast");  
    errors.add("date/time field value out of range");  
    errors.add("CHECK constraint failed");  
    errors.add("Cannot explicitly insert values into rowid column"); // TODO: don't insert into rowid  
    errors.add(" Column with name rowid does not exist!"); // currently, there doesn't seem to way to determine if  
                                                             // the table has a primary key  
    errors.add("Out of Range: Could not cast value");  
}
```

Some errors are always unexpected
(e.g., database corruptions)

Automated Testing

- ▶ Automation of test case generation and the test oracle



Generational vs. Mutational

- ▶ Mutational: start with initial seeds that are mutated
- ▶ Generational: generate inputs/test cases from scratch

(Naïve) Random Testing: Csmith

- ▶ Random program generator for C
- ▶ Challenge: generate valid programs free of undefined behavior
- ▶ Found hundreds of bugs



<https://embed.cs.utah.edu/csmith/>

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr

University of Utah, School of Computing
{xyang, chenyang, eeide, regehr}@cs.utah.edu

Abstract

Compilers should be correct. To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input. In this paper we present our compiler-testing tool and the results of our bug-hunting study. Our first contribution is to advance the state of the art in compiler testing. Unlike previous tools, Csmith generates programs that cover a large subset of C while avoiding the undefined and unspecified behaviors that would destroy its ability to automatically find wrong-code bugs. Our second contribution is a collection of qualitative and quantitative results about the bugs we have found in open-source C compilers.

Categories and Subject Descriptors D.2.5 [Software Engineering]: Testing and Debugging—testing tools; D.3.2 [Programming Languages]: Language Classifications—C; D.3.4 [Programming Languages]: Processors—compilers

General Terms Languages, Reliability

Keywords compiler testing, compiler defect, automated testing, random testing, random program generation

1. Introduction

The theory of compilation is well developed, and there are compiler frameworks in which many optimizations have been proved correct. Nevertheless, the practical art of compiler construction involves a morass of trade-offs between compilation speed, code quality, code debuggability, compiler modularity, compiler retargetability, and other goals. It should be no surprise that optimizing compilers—like all complex software systems—contain bugs.

Miscompilations often happen because optimization safety checks are inadequate, static analyses are unsound, or transformations are flawed. These bugs are out of reach for current and future automated program-verification tools because the specifications that need to be checked were never written down in a precise way, if they were written down at all. Where verification is impractical, however, other methods for improving compiler quality can succeed. This paper reports our experience in using testing to make C compilers better.

```
1 int foo (void) {  
2   signed char x = 1;  
3   unsigned char y = 255;  
4   return x > y;  
5 }
```

Figure 1. We found a bug in the version of GCC that shipped with Ubuntu Linux 8.04.1 for x86. At all optimization levels it compiles this function to return 1; the correct result is 0. The Ubuntu compiler was heavily patched; the base version of GCC did not have this bug.

We created Csmith, a randomized test-case generator that supports compiler bug-hunting using differential testing. Csmith generates a C program; a test harness then compiles the program using several compilers, runs the executables, and compares the outputs. Although this compiler-testing approach has been used before [6, 16, 23], Csmith's test-generation techniques substantially advance the state of the art by generating random programs that are expressive—containing complex code using many C language features—while also ensuring that every generated program has a single interpretation. To have a unique interpretation, a program must not execute any of the 191 kinds of undefined behavior, nor depend on any of the 52 kinds of unspecified behavior, that are described in the C99 standard.

For the past three years, we have used Csmith to discover bugs in C compilers. Our results are perhaps surprising in their extent: to date, we have found and reported more than 325 bugs in mainstream C compilers including GCC, LLVM, and commercial tools. Figure 1 shows a representative example. Every compiler that we have tested, including several that are routinely used to compile safety-critical embedded systems, has been crashed and also shown to silently miscompile valid inputs. As measured by the responses to our bug reports, the defects discovered by Csmith are important. Most of the bugs we have reported against GCC and LLVM have been fixed. Twenty-five of our reported GCC bugs have been classified as P1, the maximum, release-blocking priority for GCC defects. Our results suggest that fixed test suites—the main way that compilers are tested—are an inadequate mechanism for quality control.

We claim that Csmith is an effective bug-finding tool in part because it generates tests that explore atypical combinations of C language features. Atypical code is *not* unimportant code, however; it is simply underrepresented in fixed compiler test suites. Developers who stray outside the well-tested paths that represent a compiler's "comfort zone"—for example by writing kernel code or embedded systems code, using exotic compiler options, or automatically generating code—can encounter bugs quite frequently.

(Naïve) Random Testing: RAGS

► Random SQL query generator

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

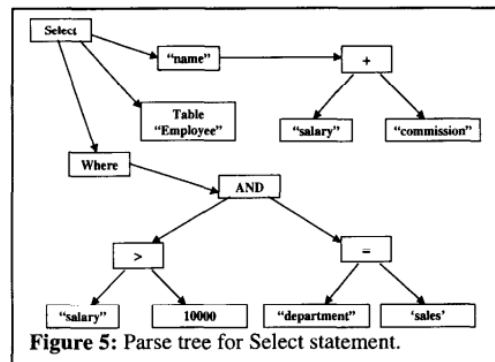


Figure 5: Parse tree for Select statement.

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.

(Naïve) Random Testing: SQLsmith

- ▶ Random SQL query generator inspired by Csmith
- ▶ Reads schema information (+ supported functions)
- ▶ Generator based on grammar
- ▶ Adopted by many/most database companies



<https://github.com/anse1/sqlsmith>

Skeletal Program Enumeration (SPE)

- ▶ Challenge: Obtain test cases with diverse control- and data-dependencies
- ▶ Idea: Extract a skeleton that can be parameterized

Skeletal Program Enumeration (SPE)

```
def bubble_sort(arr):  
    length = len(arr)  
    for i in range(length):  
        for j in range(0, length - i - 1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Skeletal Program Enumeration (SPE)

```
def bubble_sort(arr):  
    n = len(arr)  
    for i in range(length):  
        for j in range(0, length - i - 1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Variables: length, i, j

SPE describes an algorithm to **avoid** filling holes with variables that result in test cases with **redundant control- and data-dependencies**

Automated Testing & Fuzzing

Automated Testing

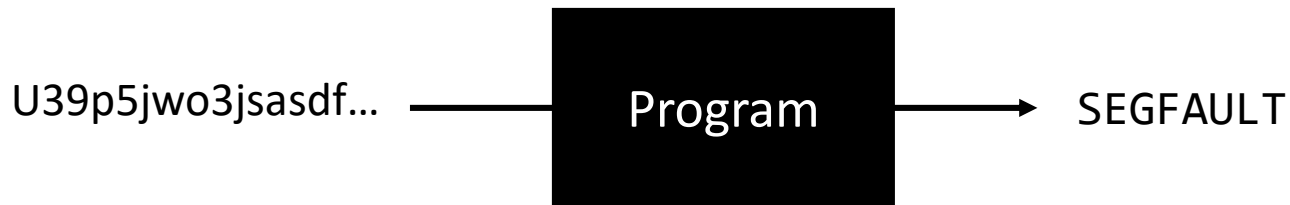
- ▶ Software engineering community
- ▶ Test own programs
- ▶ Knowledge about the domain & applications
- ▶ Test oracles

Fuzzing

- ▶ Security community
- ▶ Test other programs
- ▶ Minimal assumptions
- ▶ Security vulnerabilities

(Black-box) Fuzzing

- ▶ Bart Miller coined the term in 1988



(Black-box) Fuzzing

COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN-MADISON

CS 736
Fall 1988

Bart Miller

Project List

(Brief Description Due: Wednesday, October 26)

(Midway Interview: Friday, November 18)

(Final Report Due: Thursday, December 15)

General Comments

The projects are intended to give you an opportunity to study a particular area related to operating systems. Your project may require a test implementation, measurement study, simulation, literature search, paper design, or some combination of these.

The project suggestions below are briefly stated. They are intended to guide you into particular areas and you are expected to expand these suggestions into a full project descriptions. This gives you more freedom in selecting an area and more burden in defining your own project. There may be more issues listed for a project than you can cover. If you have a topic of your own that is not listed below, you should come and talk with me so we can work out a reasonable project description.

You will write a paper that reports on your project. This paper will be structured as if you were going to submit it to a conference. I will provide more details on the project report later in the semester.

You can work in teams of two people on the project and report.

Projects

- (1) *Operating System Utility Program Reliability – The Fuzz Generator*: The goal of this project is to evaluate the robustness of various UNIX utility programs, given an unpredictable input stream. This project has two parts. First, you will build a *fuzz* generator. This is a program that will output a random character stream. Second, you will take the fuzz generator and use it to attack as many UNIX utilities as possible, with the goal of trying to break them. For the utilities that break, you will try to determine what type of input cause the break.

The fuzz generator will generate an output stream of random characters. It will need several options to give you flexibility to test different programs. Below is the start for a list of options for features that *fuzz* will support. It is important when writing this program to use good C and UNIX style, and good structure, as we hope to distribute this program to others.

-p only the printable ASCII characters

Implicit Test Oracles

- ▶ Crashes
- ▶ Timeouts
- ▶ Dynamic Analysis Tools (Sanitizers)

Grammar Based Fuzzing/Testing

- Idea: use a grammar as a specification

```
deallocatestmt
: DEALLOCATE name
| DEALLOCATE PREPARE name
| DEALLOCATE ALL
| DEALLOCATE PREPARE ALL
;

insertstmt
: opt_with_clause INSERT INTO insert_target insert_rest opt_on_conflict returning_clause
;

insert_target
: qualified_name (AS colid)?
;

insert_rest
: selectstmt
| OVERRIDING override_kind VALUE_P selectstmt
| OPEN_PAREN insert_column_list CLOSE_PAREN (OVERRIDING override_kind VALUE_P)? selectstmt
| DEFAULT VALUES
;
```

Grey-box Fuzzing



- ▶ American Fuzzy Lop (AFL)
 - ▶ Lop, not loop
 - ▶ Successor: AFL++
- ▶ Instruments program to collect coverage
 - ▶ Grey-box fuzzing
- ▶ Puts more energy into further mutating inputs that cover new code paths

White-box Fuzzing

- ▶ Idea: utilize detailed knowledge about the program
- ▶ Symbolically execute an input
- ▶ Gather constraints along the way
- ▶ Negate constraints systematically using a constraint solver



SAGE: Whitebox Fuzzing
for Security Testing

SAGE has had a remarkable impact at Microsoft.

Patrice Godefroid, Michael Y. Levin, David Molnar, Microsoft

Most *ACM Queue* readers might think of “program verification research” as mostly theoretical with little impact on the world at large. Think again. If you are reading these lines on a PC running some form of Windows (like 93-plus percent of PC users—that is, more than a billion people), then you have been affected by this line of work—without knowing it, which is precisely the way we want it to be.

THE HIGH COST OF SECURITY BUGS

Every second Tuesday of every month, also known as “Patch Tuesday,” Microsoft releases a list of security bulletins and associated security patches to be deployed on hundreds of millions of machines worldwide. Each security bulletin costs Microsoft and its users millions of dollars. If a monthly security update costs you \$0.001 (one tenth of one cent) in just electricity or loss of productivity, then this number multiplied by a billion people is \$1 million. Of course, if malware were spreading on your machine, possibly leaking some of your private data, then that might cost you much more than \$0.001. This is why we strongly encourage you to apply those pesky security updates.

Many security vulnerabilities are a result of programming errors in code for parsing files and packets that are transmitted over the Internet. For example, Microsoft Windows includes parsers for hundreds of file formats.

If you are reading this article on a computer, then the picture shown in figure 1 is displayed on your screen after a jpg parser (typically part of your operating system) has read the image data, decoded it, created new data structures with the decoded data, and passed those to the graphics

Additional Resources

The Fuzzing Book

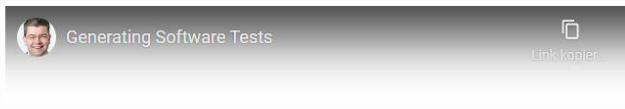
Tools and Techniques for Generating Software Tests

by Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler

About this Book

Welcome to "The Fuzzing Book"! Software has bugs, and catching bugs can involve lots of effort. This book addresses this problem by *automating* software testing, specifically by *generating tests automatically*. Recent years have seen the development of novel techniques that lead to dramatic improvements in test generation and software testing. They now are mature enough to be assembled in a book – even with executable code.

```
from bookutils import YouTubeVideo
YouTubeVideo("w4u5gCgPlmg")
```



Generating Software Tests

Breaking Software for Fun and Profit

Ansehen auf  YouTube

<https://www.fuzzingbook.org/>

Small Scope Hypothesis

► Most bugs can be found with small/simple inputs

Evaluating the “Small Scope Hypothesis”

Alexandr Andoni

Dumitru Daniliuc

Sarfraz Khurshid

Darko Marinov

MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA 02139

{andoni,dumi,khurshid,marinov}@lcs.mit.edu

Abstract

The “small scope hypothesis” argues that a high proportion of bugs can be found by testing the program for all test inputs within some small scope. In object-oriented programs, a test input is constructed from objects of different classes; a test input is within a scope of s if at most s objects of any given class appear in it. If the hypothesis holds, it follows that it is more effective to do systematic testing within a small scope than to generate fewer test inputs of a larger scope.

This paper evaluates the hypothesis for several implementations of data structures, including some from the Java Collections Framework. We measure how statement coverage, branch coverage, and rate of mutant killing vary with scope. For systematic input generation and correctness checking of Java programs, we use the Korat framework. This paper also presents the Ferastrau framework that we have developed for mutation testing of Java programs. The experimental results show that exhaustive testing within small scopes can achieve complete coverage and kill most of the mutants, even for intricate methods that manipulate complex data structures. The results also show that Korat can be used effectively to generate inputs and check correctness for these scopes.

language [12] to develop software models and exhaustively check them for small scopes with the Alloy Analyzer. These case studies showed that the hypothesis holds for those software models, but they did not consider the actual implementations.

This paper evaluates the “small scope hypothesis” for several benchmark programs, including some data structure implementations from the Java Collections Framework [29]. Evaluating the hypothesis requires determining the scope up to which each program should be checked, i.e., the sufficient scope that gives significant confidence that the program has no bugs. We use code coverage and mutation testing criteria to determine the sufficient scope.

Code coverage is a common criterion for assessing the quality of a set of test inputs [5]. Measuring code coverage involves executing the program on each input and recording parts of the program (e.g., statements, branches, paths) that get executed. Statement (branch) coverage is then the ratio of the number of executed statements (branches) to the number of total statements (branches) in the program; *complete coverage* is the ratio of 100%.

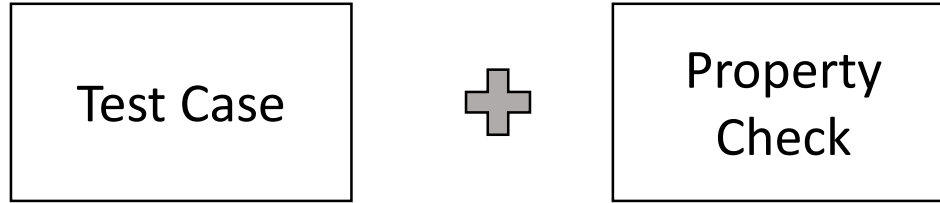
Mutation testing is another criterion for assessing the quality of a set of test inputs [11, 26]. Mutation testing proceeds in two steps. In the first step, several *mutants* are generated from the original (correct) program, by performing one or more syntactic modifications. These modifications are specified by *mutation operators*, e.g., replacing a vari-

Test Case Reduction

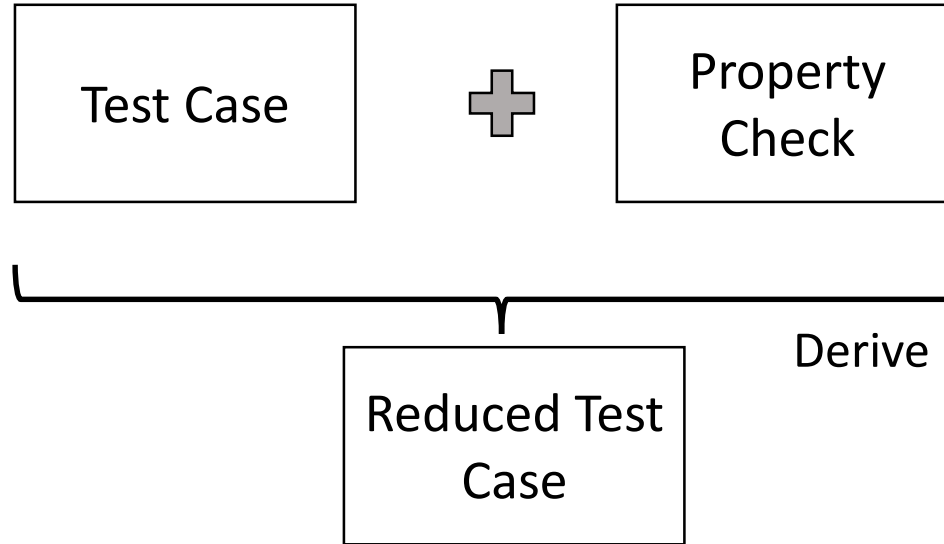
- ▶ Problem: Automatically generated test cases are typically unnecessarily large
- ▶ Idea: reduce them

```
PRAGMA cache_size = 50000;
PRAGMA temp_store=MEMORY;
PRAGMA synchronous=off;
PRAGMA encoding = 'UTF-16be';
CREATE TABLE IF NOT EXISTS t0 (c0 TEXT , c1 INTEGER );
INSERT OR IGNORE INTO t0(c1) VALUES (320), (0X2a1);
PRAGMA incremental_vacuum;
UPDATE OR ABORT t0 SET c1 = '-9223372036854775808' WHERE t0.c0 COLLATE BINARY;
INSERT OR IGNORE INTO t0 VALUES (0x58, 0x2c4), (852, 0X3d4), (384, 0x199), (718, 847);
BEGIN TRANSACTION;
PRAGMA legacy_file_format = true;
UPDATE OR REPLACE t0 SET c0 = 0.03283042105908274, c0 = x", c1 = 0Xffffffab13f8ae;
UPDATE t0 SET (c0)=(-1881505887) WHERE ((t0.c1 COLLATE NOCASE)IS NOT(((t0.c0) NOT BETWEEN (t0.c1
PRAGMA auto_vacuum;
UPDATE OR IGNORE t0 SET c0 = 'zlvRGBI 2vx?öd ☹️', c1 = -1.424754514E9, c0 = 0.3406202076523681;
UPDATE OR ABORT t0 SET c1 = 0.6840655763518566 WHERE (-45384822 IN ());
UPDATE OR IGNORE t0 SET c0 = 'Xf', c0 = 0Xffffff8fda7fa1 WHERE (~ ((NOT (t0.c0)))));
BEGIN TRANSACTION;
PRAGMA temp.auto_vacuum;
BEGIN TRANSACTION;
ANALYZE;
UPDATE OR REPLACE t0 SET (c0)=(0.06651996418720685);
REINDEX;
REINDEX BINARY;
REINDEX;
COMMIT TRANSACTION;
PRAGMA main.mmap_size;
CREATE INDEX IF NOT EXISTS i48 ON t0((CASE c1 WHEN 0.36391525195019603 THEN c0 WHEN c0 THEN c0
WHEN c0 THEN c1 ELSE 0.19854138768397123 END IN (((c1)&(c0)))) ASC,CAST(c1 COLLATE RTRIM AS BLOC
DESC);
COMMIT;
ANALYZE;
ROLLBACK TRANSACTION;
COMMIT;
PRAGMA cache_size;
ANALYZE;
BEGIN DEFERRED TRANSACTION;
PRAGMA journal_mode = WAL;
UPDATE OR REPLACE t0 SET c0 = 'j(!!c
R 獠>H趨|/FZ]>!', c0 = 0.517704345533682, c0 = NULL;
PRAGMA busy_timeout;
COMMIT;
COMMIT TRANSACTION;
PRAGMA main.soft_heap_limit;
PRAGMA temp.threads = 5428778230116844349;
PRAGMA temp.reverse_unordered_selects = false;
PRAGMA main.journal_mode;
COMMIT TRANSACTION;
```


Test Case Reduction



Test Case Reduction



Reduction Tools

```
extern int printf (const char *, ...);
static char (safe_1) (char si)
{

    return (si == (-128)) ? ((si)) : -si;
}

static char (safe_2) (char si1, char si2)
{

    return
        (((si1 > 0) && (si2 > 0) && (si1 > ((127) - si2)))
        || ((si1 < 0) && (si2 < 0)
            && (si1 < ((-128) - si2)))) ? ((si1)) : (si1 +
si2);
}
...
```

Original test case

```
#!/bin/bash
grep goto small.c >/dev/null 2>&1
```

Interestingness test

C-Reduce

goto

Reduced test case

Reduction Tools

- ▶ Delta Debugging
- ▶ Perses
- ▶ SQL-Reduce
- ▶ ...

Yesterday, My Program Worked.
Today, It Does Not. Why?

Andreas Zeller

Universität Passau
Lehrstuhl für Software-Systeme
Innstraße 33, D-94032 Passau, Germany
zeller@acm.org

Abstract. Imagine some program and a number of changes. If none of these changes is applied (“yesterday”), the program works. If all changes are applied (“today”), the program does not work. Which change is responsible for the failure? We present an efficient algorithm that determines the minimal set of failure-inducing changes. Our *delta debugging* prototype tracked down a single failure-inducing change from 178,000 changed GDB lines within a few hours.

1 A True Story

The GDB people have done it again. The new release 4.17 of the GNU debugger [6] brings several new features, languages, and platforms, but for some reason, it no longer integrates properly with my graphical front-end DDD [10]: the arguments specified within DDD are not passed to the debugged program. Something has changed within GDB such that it no longer works for me. Something? Between the 4.16 and 4.17 releases, no less than 178,000 lines have changed. How can I isolate the change that caused the failure and make GDB work again?

Test Case Deduplication

- ▶ Problem: testing tools might repeatedly generate bug-inducing test cases for the same underlying bugs
- ▶ Idea: Assign test cases that trigger the same bug to different buckets
- ▶ AFL deduplicates bugs based on the stack trace

Test Case Deduplication

- ▶ Problem: testing tools might repeatedly generate bug-inducing test cases for the same underlying bugs
- ▶ Idea: Assign test cases that trigger the same bug to different buckets
- ▶ AFL deduplicates bugs based on the stack trace

Related Topics

- ▶ Static Analysis
- ▶ Dynamic Analysis
- ▶ Program Repair
- ▶ Runtime Verification
- ▶ Symbolic Execution
- ▶ Abstract Interpretation
- ▶ Model Checking
- ▶ Program Synthesis
- ▶ Compilation & Optimization
- ▶ ...

Static vs. Dynamic Analysis

Static Analysis

- ▶ Examines the code
- ▶ Considers all possible executions (or none)
- ▶ Often tension between finding all bugs and reporting only true bugs

Dynamic Analysis

- ▶ Analyzes the program while it is running
- ▶ Typically considers a single execution
- ▶ Typically no false alarms → every bug reported indicates a real bug

Soundness and Completeness

What Does It Mean for a Program Analysis to Be Sound?

by Ilya Sergey on Aug 7, 2019 | Tags: abstract interpretation, concurrency, dynamic analysis, soundness, static analysis, testing



Additional Resources

The Debugging Book

Tools and Techniques for Automated Software Debugging

by Andreas Zeller


About this Book

Welcome to "The Debugging Book"! Software has bugs, and finding bugs can involve lots of effort. This book addresses this problem by *automating* software debugging, specifically by *locating errors and their causes automatically*. Recent years have seen the development of novel techniques that lead to dramatic improvements in automated software debugging. They now are mature enough to be assembled in a book - even with executable code.

```
from bookutils import YouTubeVideo
YouTubeVideo("-n0xI6Ev_I4")
```



The Debugging Book

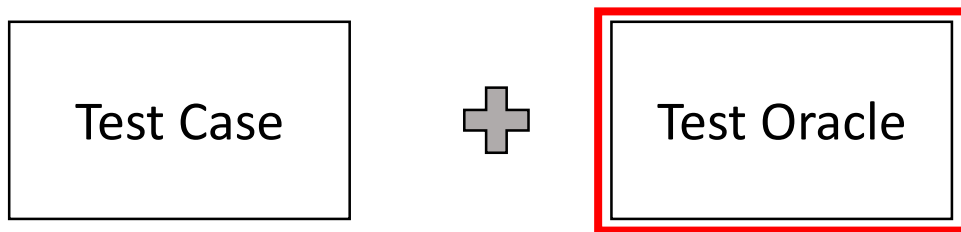
Ansehen auf  YouTube

debuggingbook.org
Music: bensound.com

<https://www.debuggingbook.org/>

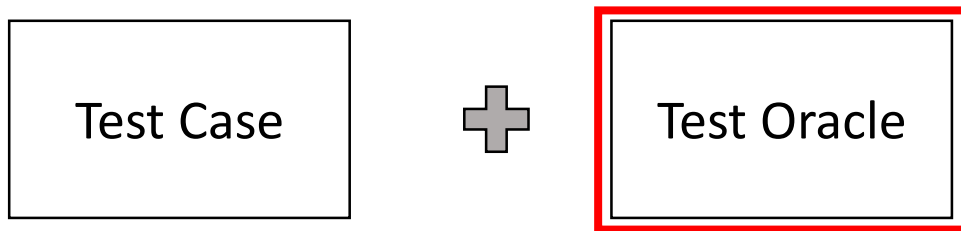
Presentations: Differential Testing

- ▶ Data-Oriented Differential Testing of Object-Relational Mapping Systems
- ▶ APOLLO: automatic detection and diagnosis of performance regressions in database systems



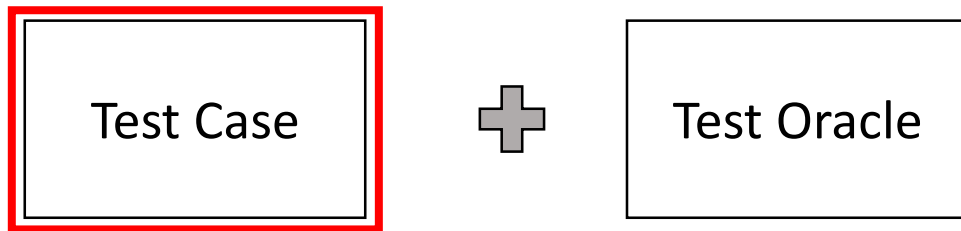
Presentations: Metamorphic Testing

- ▶ Metamorphic testing of Datalog engines
- ▶ Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries



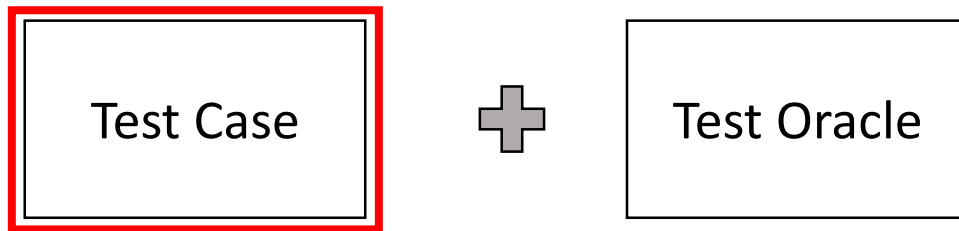
Presentations: Fuzzing

- ▶ SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback
- ▶ BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction



Presentations: Fuzzing

- ▶ Search-based test data generation for SQL queries
- ▶ Data generation for testing and grading SQL queries
 - ▶ Mutation testing technique



Presentations: Testing and Synthesis

- ▶ **Torturing Databases for Fun and Profit**
 - ▶ Property-based testing
 - ▶ Systems level approach
- ▶ **Synthesizing Analytical SQL Queries from Computation Demonstration**
 - ▶ Program synthesis

Presentations: Analysis and Debugging

- ▶ CheckCell: data debugging for spreadsheets
- ▶ SQLCheck: Automated Detection and Diagnosis of SQL Anti-Patterns

Presentations: Performance Optimization

- ▶ View-Centric Performance Optimization for Database-Backed Web Applications
- ▶ AIDA - Abstraction for Advanced In-Database Analytics

Summary

- ▶ Manual/unit testing, code coverage, mutation testing
- ▶ Test oracles: Differential testing, metamorphic testing, property-based testing, intramorphic testing
- ▶ Test case generation: generational vs. mutational, automated testing vs fuzzing
- ▶ Test case reduction and deduplication